

WL-TR-95-1120

**AVIONICS SOFTWARE REENGINEERING
TECHNOLOGY (ASRET) PROJECT
VOLUME 2
Reengineering Tool (RET) Diagrams**

D.E. WILKENING



TASC

55 Walkers Brook Drive
Reading, Massachusetts 01867

MAY 1995

Project Final Report for 5/1/92 – 5/1/95

Approved for public release; distribution is unlimited.

19960325 035

**AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-7409**

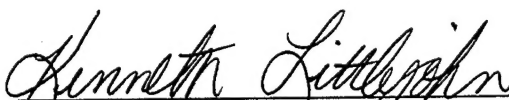
DTIC QUALITY INSPECTION 6


NOTICE


When Government drawings, specifications or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.


KENNETH LITTLEJOHN, Project Engineer
Software Concepts Section
WL/AAAF-2


WILLIAM R. BAKER, Acting Chief
Avionics Logistics Branch
WL/AAAF


STEPHEN G. PETERS, Lt Col, USAF
Deputy Chief
System Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization, please notify WL/AAAF, WPAFB, OH 45433-7301 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average one hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE May 1995		3. REPORT TYPE AND DATES COVERED Final 5/1/92 – 5/1/95	
4. TITLE AND SUBTITLE Avionics Software Reengineering Technology (ASRET) Project, Volume 2 Reengineering Tool (RET) Diagrams				5. FUNDING NUMBERS C F33615-92-D-1052 PE 78012 PR 3090 TA 01 WU 14	
6. AUTHOR(S) D.E. Wilkening (TASC)					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) TASC 55 Walkers Brook Drive Reading, MA 01867				8. PERFORMING ORGANIZATION REPORT NUMBER TASC: TR-06661-5	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Avionics Directorate Wright Laboratory Air Force Materiel Command Wright-Patterson AFB, Ohio 45433-7409				10. SPONSORING/MONITORING AGENCY REPORT NUMBER WL-TR-95-1120	
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This document presents results of the Avionics Software Reengineering Technology (ASRET) project. It provides samples of output that we created using the Reengineering Tool (RET) prototype that we developed under ASRET, narrates our efforts to produce the output, and reports on insights that we gained in the process. A companion document, <i>Volume I, Project Summary, Account, and Results</i> (Ref. 1), presents the findings of the ASRET Project and describes the RET prototype and the context within which we produced the output.					
14. SUBJECT TERMS Reengineering, Reverse Engineering, Reuse				15. NUMBER OF PAGES 174	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL		

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

TABLE OF CONTENTS

Page

LIST OF TABLES		iv
1. INTRODUCTION		1
1.1 Background		1
1.2 ASRET Project Overview		2
1.2.1 Project Goals		2
1.2.2 Task Structure		2
1.3 Report Organization		3
2. RET VIEWS		4
2.1 FCR Subsystem		6
2.2 GPS Subsystem		9
2.3 INS Subsystem		11
2.4 MFD Subsystem		14
2.5 RLT Subsystem		14
2.6 UFC Subsystem		17
2.7 SMS Subsystem		18
REFERENCES		19
APPENDIX		
A Exhibit FCR-P		20
B Exhibit FCR-D		29
C Exhibit GPS-P		48
D Exhibit GPS-D		54
E Exhibit INS-P		69
F Exhibit INS-D		82
G Exhibit MFD-P		88
H Exhibit MFD-D		98
I Exhibit RLT-P		133
J Exhibit RLT-D		140
K Exhibit UFC-P		144
L Exhibit UFC-D		153

LIST OF TABLES

Table		Page
1	ASRET Task Structure	2
2	Block 40 Subsystems	4

1.

INTRODUCTION

This document presents results of the Avionics Software Reengineering Technology (ASRET) project. It provides samples of output that we created using the Reengineering Tool (RET) prototype that we developed under ASRET, narrates our efforts to produce the output, and reports on insights that we gained in the process.

A companion document, *Volume I, Project Summary, Account, and Results* (Ref. 1), presents the findings of the ASRET Project and describes the RET prototype and the context within which we produced the output.

1.1 BACKGROUND

Context — The Avionics Software Technology Support (ASTS) program is an ongoing activity of the Software Concepts Group, Avionics Logistics Branch at Wright Laboratory (WL/AAAF-3), Wright Patterson Air Force Base, Ohio. The objective of the ASTS program is to perform research and development for enhancing Embedded Computer System (ECS) software development and postdeployment support.

The ASRET project is the second delivery order (DO) issued to TASC under the ASTS program. The main objective of ASRET was to develop an automated Reengineering Tool (RET) prototype for avionics support software. The specific goals included investigating existing reengineering and reverse engineering processes, techniques, and software tools, defining a reengineering process model, and building the RET prototype software tool.

Rationale — The reengineering of software from one language to another is becoming a necessity as Air Force organizations strive to modernize and improve the maintainability of their systems while avoiding the excessive costs of new development. Systems that have been in use for years often incur large maintenance costs (Ref. 2) for a number of reasons:

- Continual maintenance has made the current implementation and original design inconsistent, made the code harder to understand and error-prone, and made the documentation out-of-date.
- They are written in languages that have fallen out of favor. The limited selection of support tools for these languages, the corresponding expense of these tools, and the shrinking pool of qualified programmers to maintain the software adds to the expense of maintenance.

- They were developed without modern software engineering practices, resulting in code that lacks structure and is difficult to understand.
- Employee turnover has reduced the staff's understanding and "intimate" knowledge of the system.

Wright Laboratory initiated the ASRET project to help reduce maintenance costs for legacy systems and to assist in the evolution to Ada. To this end, we developed an environment for reengineering software from one language to another. We concentrated on the reengineering of avionics simulation software written in FORTRAN to Ada, and designed the RET so that additional languages could be supported in the future.

1.2 ASRET PROJECT OVERVIEW

1.2.1 Project Goals

The objective of the ASRET project was to develop an automated Reengineering Tool (RET) prototype for avionics support software. The specific goals included investigating existing reengineering and reverse engineering processes, techniques, and software tools, defining a reengineering process model, and building a RET prototype that supports:

- Translating avionics simulation software written in FORTRAN to Ada
- Improving the software through restructuring techniques
- Changing the design of the software so that it is consistent with modern software engineering principles
- Adding documentation that is consistent with the software.

1.2.2 Task Structure

The ASRET project was structured as shown in Table 1.

Table 1 ASRET Task Structure

NUMBER	NAME
1	Software Reengineering Study
2	Reengineering Process Model Development
3	Reengineering Tool Development
4	Reengineering Tool Testing and Evaluation

For Task 1, *Software Reengineering Study*, we conducted an extensive investigation of software reengineering tools and methods. The goal was to collect, organize, and present information on software reengineering tools and methods that might be relevant to ASRET, and to record the information for use in the subsequent tasks. The results of Task 1 are documented in the Software Reengineering Study Report (Ref. 3).

During Task 2, *Reengineering Process Model Development*, we developed a reengineering process model based upon the results of Task 1, and developed the Software Requirements Specification (Ref. 4) for the Reengineering Tool (RET) prototype.

In Task 3, *Reengineering Tool Development*, we designed (Ref. 5) and implemented the RET prototype and exercised the RET by transforming the Fire Control Radar (FCR) subsystem of the F-16 Operational Flight Program (OFP) simulation software provided to us by Wright Laboratory for this purpose.

We executed the RET prototype and converted most of the other subsystems from Block 40 of the F-16 OFP simulation software to Ada in Task 4, *Reengineering Tool Testing and Evaluation*.

1.3 REPORT ORGANIZATION

The Avionics Software Reengineering Technology (ASRET) Project Final Report comprises two volumes. This document, *Volume II, Reengineering Tool (RET) Diagrams*, provides sample hardcopy that we produced with the RET. Section 1 describes the ASRET Project and provides background information. Section 2 describes each of the diagrams that we created during the Reengineering Tool Testing and Evaluation task. The exhibits contain the Packager and Dataflow Diagrams.

A companion document, *Volume I, Project Summary, Account, and Results* (Ref. 1), presents the findings of the ASRET Project and describes the RET prototype and the context within which we produced the diagrams in Vol. II.

2.

RET VIEWS

This section describes the views provided in the exhibits. We discuss the activities that we performed under the Reengineering Tool Testing and Evaluation task (Task 4) to create the views and some insights that we gained during the exercise.

During Task 4, we analyzed six subsystems from Block 40 of the F-16 OFP simulation software provided to us by WL/AAAF-3. Table 2 names the subsystems and lists the sections and corresponding exhibits containing the graphs. We did not analyze the SMS subsystem so this document doesn't provide a corresponding exhibit.

Identifiers for individual graphs in the exhibits take the form SSS-V-n, where SSS denotes the subsystem, V is P for Packager or D for DFD, and n enumerates the graphs within each subsystem's view. For example, FCR-P-3 identifies the third graph in the Packager view for the FCR subsystem.

Table 2 Block 40 Subsystems

SUBSYSTEM	SECTION	EXHIBIT	
		PACKAGER	DFD
FCR	2.1	FCR-P	FCR-D
GPS	2.2	GPS-P	GPS-D
INS	2.3	INS-P	INS-D
MFD	2.4	MFD-P	MFD-D
RLT	2.5	RLT-P	RLT-D
UFC	2.6	UFC-P	UFC-D
SMS	2.7	none	none

Volume II does not explain many of the terms defined in Vol. I (Ref. 1). It assumes that the reader is familiar with the Packager and Dataflow Diagram views described in Vol. I. For each subsystem, we discuss only the graphs that illustrate our activities and the insights that we gained during Task 4. We do not attempt to describe every graph in detail.

Despite our efforts to focus only on the most important portions of the graphs, the narrative is sometimes dense. We provide the details because examples are the most effective way to convey much of the information. Vol. I summarizes the information, but the examples in Vol. II substantiate our statements.

We designed the RET prototype views for an interactive environment. We did not intend them to be used in hardcopy form. The interactive views provide information that may not appear on a sample printout. The views allow the engineer to filter the information and control the degree of detail shown. For example, the edge labels in the Packager and DFD views each provide three different levels of detail, but only one may appear in a given printout.

The engineer forms a composite understanding of the information in the views by alternating among various perspectives and controlling what information is shown. The hardcopy samples do not convey the depth and variety of information available to the engineer. We present them as a means of explaining features of the RET prototype rather than to document the Block 40 code.

The process by which we created the hardcopy demanded more attention to the technical details of interacting with the RET prototype than we would have liked. For example, we arranged the graphs by dragging nodes with the mouse because the RET prototype doesn't automatically arrange them. We focused on implementing reengineering methods and techniques rather than on the prototype's user interface. We assume that a production version of the RET would address the user interface and other usability issues, as we recommend in Vol. I.

The impact of this on our efforts to produce the graphs was that most of our attention was initially absorbed in the *process* of creating the diagrams as opposed to studying their *content*. We assessed the diagrams during a subsequent review. This highlights the importance of addressing the usability issues of the RET to reduce distractions during the reengineering process. Our experience also validates the iterative top-down approach recommended by the ASRET process model (described in Vol. I).

2.1 FCR SUBSYSTEM

Packager — Exhibit FCR-P contains samples of the Packager view for the FCR subsystem. FCR-P-1 depicts five packages as nodes or rectangles and the data binding relationships among them as thin undirected edges or lines. The package “dead code” is so named because we initially found no calls to any of its subprograms, but the edges with “fcr_df” and “modes” show that the package is not isolated. The graph does not show whether any of the packages are referenced from outside the FCR subsystem.

FCR-P-2 shows the five subprograms in “dead code.” We see that they do not share any data bindings. FCR-P-3 shows the nineteen subprograms in the package “main.” The edges show the interconnection strength between the subprograms. Many of the edges show only one data binding, but due to our definition of data binding, a value of one may indicate zero bindings and one subprogram call.

This ambiguity is the main reason that we recommend reversing our decision to add one to the interconnection strength for subprogram calls. In an interactive display, we could click on any edge to expand the label and determine if a subprogram call is involved, but we can’t tell from the hardcopy. We have found in the FCR code, however, that most edges with an interconnection strength of one indicate subprogram calls.

Although the subprograms are highly interrelated, most of the edges have very low interconnection strengths. The highest value is four, and most of the edges are less than three. FCR-P-4 through FCR-P-7 show the subprograms in the package “modes.” The first two show only select edges, but the last two graphs are hopelessly cluttered. They show only that the subprograms in “modes” are very highly interrelated.

FCR-P-8 shows the fifteen subprograms in the package “fcr_dr.” The interconnection strength values shown on the edges imply that the edges represent subprogram calls rather than data bindings. While interacting with the first graph, we determined that the subprogram “fcr_output” is called from “main” and that the subprograms named “fcr_drxxx” are called from “modes.” This is not shown in the sample hardcopy.

We address the general difficulties of managing the kind of complexity apparent in FCR-P-3 through FCR-P-7 at various points below with similar examples drawn from other subsystems, primarily the UFC (Section 2.6). This challenge is ubiquitous in reengineering large systems and we will discuss several ways that the RET prototype combats the problem in both the Packager and DFD views.

DFD — Exhibit FCR-D contains samples of the DFD view for the FCR subsystem. The top level graph (FCR-D-1) of the DFD shows transform nodes as ellipses and buffer nodes as rectangles. The transform nodes correspond to subprograms. The buffer nodes correspond to global variables. Most of the subprograms in the FCR subsystem don't have arguments, so there are no buffers that correspond to formal parameters. The arrows in the graph indicate the direction of dataflow between the transform and buffer nodes.

The transform nodes in FCR-D-1 represent subprograms that are not called by any other subprogram in the FCR subsystem. They are probably called by other subsystems. We didn't realize this when we created the Packager view which is why four of the subprograms (FCR_TERM, FCR_SUSPEND, SET_SCALED_MUX, and READ_SCALED_MUX) appear in the "dead code" package in the Packager view.

We did not go back to fix the package structure because we were only performing an evaluation of the RET prototype. In an actual reengineering project, the engineer would take advantage of insights gained by examining the DFD and fix the structure through the Packager view on the next iteration in accordance with the ASRET process model (Ref. 1).

The rectangle labeled "DR*" is a buffer collection that contains a set of buffer nodes representing variables that begin with "DR." In the interactive view, the engineer may click on the node to show the buffers nested under the collection. Buffer collection nodes are identified by the presence of asterisks in their labels.

The label of the buffer collection node is determined by the buffer mapping that we coded for the FCR subsystem. We defined a mapping for the FCR code that takes any buffer that begins with "DR" to "DR*." We observed the convention that an asterisk represents zero or more characters, but the engineer is free to provide an arbitrary mapping; the domain and range of the mapping are sets of strings that the engineer specifies.

The bidirectional arrow between DR* and FCR_OUTPUT shows that the latter reads and writes variables with names that begin with "DR." The arrow from the buffer node directly below FCR_OUTPUT indicates that the latter reads variables IIFCRQ and IFCRRQ, but doesn't write them. The arrow to the buffer node on the right of FCR_OUTPUT shows that FCR_OUTPUT writes the variable named FCR_IFILL, but doesn't read it.

The large buffer node on the far left contains one buffer named "FPSKTS" and six collections. The buffer collection node to the immediate right contains 32 buffers. The label begins with the integer because the number of nested buffers exceeds the predefined threshold. The node is not large enough to display all the nested buffer names so the label is clipped after the first buffer node name, "IS17J."

FCR-D-2 shows that FCR_OUTPUT calls 11 subprograms. Terminal transform nodes named FCR_DRnnn, where nnn denotes various integers between 003 and 033, represent nine of the subprograms. The presence of the FCR_DRnnn nodes in the graph implies the existence of the subprogram calls, by definition of the DFD. The graph clearly shows that the subprograms read variables named MDR* and write variables named DR*.

The nodes that represent subprograms FCR_SYMBOLS and FCR_ADO are also terminal transform nodes. The body transform node labeled FCR_OUTPUT represents the statements in the body of subprogram FCR_OUTPUT, by definition of the DFD. The arrow to that node in the graph shows that the statements read variables named DR*.

The DFD does not visually distinguish the three kinds of transform nodes (body, terminal, and nonterminal) due to limitations of INTERVISTA. We know that FCR_OUTPUT is a *body* transform node because it appears in the graph for FCR_OUTPUT. We know that the nodes named FCR_DRnnn are nonterminal transform nodes because the DFD provides graphs for them, but the engineer can only determine that fact by interacting with the view (clicking on the nodes with the mouse to bring up the graphs). Nonterminal transform nodes represent subprograms that call other subprograms.

FCR-D-3 shows that FCR_INPUT calls subprograms named FCR_DFnnn that read variables named DF* and write variables named MDF*. The structure is similar to that of FCR_OUTPUT. Apparently, FCR_INPUT transfers data from DF* to MDF* via the FCR_DFnnn subprograms whereas FCR_OUTPUT transfers data from MDR* to DR* via the FCR_DRnnn subprograms.

The dataflow between MDR* and DR* through FCR_OUTPUT is shown in FCR-D-1, but the dataflow between MDF* and DF* through FCR_INPUT isn't prominent. The buffer for MDF* is hidden in the buffer collection node that shows 32 in the label, but we can't see it unless we examine the graph for the collection. The buffer for DF* does appear in one of the buffer collections in the first graph, but it's inconspicuous.

This is one reason that we would like to allow the engineer to edit the DFD. In this case, the engineer might highlight the structural similarity between FCR_INPUT and FCR_OUTPUT by bringing the MDF* buffer to the top graph and moving the DF* buffer to its own node on that graph.

FCR-D-4 shows the dataflow within FCR_DF033. The graph points out an anomaly in the DFD. The graph should show an arrow from READ_MUX to DF033. The RET did not correctly interpret the dataflow across the parameter chain, in this case, because the parameter is an element of an array. The same problem shows up in the graph for FCR_DF031 (FCR-D-5). This is an example of a problem that must be corrected before applying the RET to other applications. FCR-D-6 and FCR-D-7 represent relatively simple subprograms.

The graph for subprogram FCR (FCR-D-8) typifies the degenerate case of an overly complicated DFD. We include a version (FCR-D-8) in which the edges are hidden and one (FCR-D-9) that shows all edges. About all that the engineer can tell from these graphs is that FCR calls many subprograms and references many variables. The graphs for FCRMOD (FCR-D-12 through FCR-D-15) and FCRS16 (FCR-D-10 through FCR-D-11) exhibit the same problem. FCR-D-16 through FCR-D-18 are much simpler.

2.2 GPS SUBSYSTEM

Packager — Exhibit GPS-P contains samples of the Packager view for the GPS subsystem. GPS-P-1 shows four packages that contain subprograms from the GPS subsystem; their names are shown in lower-case. The generated packages are on the left. We know from examining the interactive views that subprograms GPS_PROCESS, GPS_INPUT, and GPS_OUTPUT are the primary subprograms that are called from outside the GPS subsystem.

GPS-P-2 through GPS-P-4 show the packager views for packages gps_interface, gps_process_gp, and gps_process_fc, respectively. We learned from the DFD (after we had created this Packager view) that GPS_INPUT calls subprograms in gps_process_fc, and GPS_OUTPUT calls subprograms in gps_process_gp. After looking at the DFD, we think that either GPS_OUTPUT belongs in gps_process_gp, or GPS_INPUT belongs in gps_interface. We would need to examine the source code views to decide which alternative is better, but we can see that they are treated asymmetrically in this Packager view, and this suggests to us that we must investigate further.

Our first thought was to place all subprograms that are called from outside of GPS in package `gps_interface`. This results in a relatively large number of data bindings between the packages and a very small number of bindings among the subprograms within any given package. For example, the edges in GPS-P-2 through GPS-P-4 show few, if any bindings. GPS-P-5 shows what may be utility subprograms.

We see in GPS-P-1 that `gps_interface` shares a relatively great number of data bindings with `gps_process_gp`. Out of the 37 bindings 29 are due to `GPS_INIT` and 12 (possibly overlapping) bindings are with `GPSINT`. Both subprograms are related to initialization. We are unconcerned by the number of bindings because we would expect initialization routines to affect many variables. We can't see the numbers in the hardcopy, but we can display them on the label of the edge between the two packages in the interactive view.

DFD — Exhibit GPS-D contains samples of the DFD view for the GPS subsystem. GPS-D-1 shows the variables that are used by the input and output subprograms. It shows the subprograms (as transform nodes) that are not called from within the GPS subsystem. It shows, for example, that subprograms `PROCESS_GPS00n`, where `n` is 7, 8, and 9, are not called from within GPS. (They may be called from outside the GPS subsystem, but the DFD provides no information on this.)

GPS-D-2 shows that subprograms `PROCESS_GPS00n`, where `n` is 1, 2, 3, 5, and 6, are called by `GPS_OUTPUT`. This asymmetry for the subprograms `PROCESS_GPS00n` piqued our curiosity, so we examined the Source Code Listings by clicking on the transform nodes. We found that all statements in `PROCESS_GPS007` and `PROCESS_GPS008` are commented-out.

Comments in each `PROCESS_GPS00n` state that "this subroutine is called by `GPS_OUTPUT` to pack data for Mux messate `GP00n`." `PROCESS_GPS009` doesn't appear in the graph for `GPS_OUTPUT` and when we looked at the Source Code Listing for `GPS_OUTPUT`, we found that the call to `PROCESS_GPS009` is commented out. The subprograms named `PROCESS_GPS00n` that appear in the `GPS_OUTPUT` graph (GPS-D-2) do contain numerous statements, suggesting that they are still in use.

GPS-D-3 through GPS-D-7 show the details of the `PROCESS_GPS00n` subprograms that are still in use. The graph for `PROCESS_GPS001` (GPS-D-3) shows which variables it outputs through `SET_MUX`. The graph doesn't show that it implements an output function, but we know that it is called from within `GPS_OUTPUT`. The arrows in

the graph are bidirectional because all formal parameters in the generated Ada code have mode "in out." When the Transformer is enhanced to generate more discriminating formal parameter modes, the arrows in the DFD that correspond to mode "in" and mode "out" parameters will be unidirectional. The implication for GPS-D-3 is that the direction of dataflow would be more accurate.

GPS-D-8 through GPS-D-10 show a structure for GPS_INPUT that is analogous to that of GPS_OUTPUT. The graphs for GPS_COMPUTE (GPS-D-11) and GPSNAV (GPS-D-12) show how GPSNAV, GPSTMR, GPSINT, and GPSERR are local to the gps_interface package in the sense that they are within the calling scope of GPS_COMPUTE. We discussed moving GPS_OUTPUT out of that package above, and it now looks like the gps_interface package might be more appropriately named the gps_compute package. GPS-D-13 shows subprogram GPS_INIT.

When we get to the graph for GPSBIT (GPS-D-14), we become suspicious that perhaps it isn't really called from outside the GPS subsystem, as might be implied by its appearance on the top-level graph (GPS-D-1). Its name doesn't fit the pattern for such externally-called subprograms that we feel is emerging. After looking at the GPS_COMPUTE structure in GPS-D-11, we wonder why GPSBIT isn't present, since other similarly-named GPSxxx subprograms are declared with GPS_COMPUTE in the gps_interface package.

When we inspect the Source Code Listing for GPSBIT, we find a comment ("ZER-OUT - GPS OUTPUT BLANKING ROUTINE") that leads us to think it should be in the GPS_OUTPUT structure. We inspect the Source Code Listing for GPS_OUTPUT and find that all calls to GPSBIT are commented-out. That is why it appears on the top-level DFD graph (GPS-D-1). We deduce that it is dead code.

If we could delete the nodes representing dead code that we found in the top-level DFD graph (GPS-D-1), the graph would be less complicated. This is one reason that we recommend additional editing capabilities for the DFD. In any case, we would remove the dead code from the system during the next iteration through the Packager view.

2.3 INS SUBSYSTEM

Packager — Exhibit INS-P contains samples of the Packager view for the INS subsystem. INS-P-1 through INS-P-5 show the library units generated by the Packager (upper-case) and the packages that we formed by clustering (lower-case). INS-P-1 shows

the number of data bindings on each edge. INS-P-2 shows the calling relations. INS-P-3 shows which subprograms are involved in the calls, but we can't see which calls which. We can infer this from the direction of the arrows for all but one, the bidirectional arrow between `ins_utilities` and `ins_processing`. INS-P-4 explicitly shows the individual calls, but doesn't show which package the subprograms are declared in. The information on the arrows in INS-P-3 and INS-P-4 are complementary.

The arrow between `ins_utilities` and `ins_processing` is interesting because it shows that calls are made in both directions. We would prefer a layered calling structure for the packages to reduce the effects of dependencies during maintenance and because a layered structure is easier to understand. INS-P-6 and INS-P-7 show the subprograms in `ins_processing` and `ins_utilities`, respectively.

We separated the subprograms into the two packages based upon their names (all subprograms in `ins_processing` begin with `INS`), and the fact that only the subprograms in `ins_utilities` share data bindings. These are two very weak reasons for clustering. The subsystem might be better structured if these two packages were combined. We also tried to minimize the number of data bindings between these two packages, and also the `ins_io_processing` package (INS-P-8), during clustering. To improve the package structure beyond what we have developed requires examining the Source Code Listings, but we did not go to that level of detail for this subsystem during the RET prototype evaluation.

INS-P-9 shows the `ins_interface_routines` package. We placed subprograms that appear to be called from outside the `INS` subsystem into this package.

We applied several techniques to form the `ins_ad_io`, `ins_fcc_io`, and `ins_model_io` packages, INS-P-10 through INS-P-12, respectively. Clustering by common clients and suppliers revealed that the subprograms in these packages are the only ones called by `INS_INPUT` and `INS_OUTPUT`. This can also be seen in the library-level graphs (INS-P-1). We differentiated the packages based on the similarity of the names of the subprograms and our examination of comments in the Source Code Listings. Based on the comments, we added the parenthetical information in the labels that helps distinguish the purpose of each subprogram in the `io` packages.

DFD — Exhibit INS-D contains samples of the DFD view for the `INS` subsystem. The top-level graph (INS-D-1) is very busy due to the number of buffers and problems clipping their labels. With so much information in the graph, we can make only general observations. The transformations represent subprograms that are not called from within

the INS subsystem. They correspond with the subprograms that we placed in the `ins_interface_routines` package (INS-P-9).

INS-D-2 shows details on the `INS_INIT` subprogram. We are not surprised that an initialization subprogram writes sixty-seven variables. The `INS_INPUT` (INS-D-3) and `INS_OUTPUT` (INS-D-4) graphs show the structure of the subprograms. The dataflow would be much improved if the code generator generated the appropriate formal parameter modes. The graphs show the subprograms that the input and output routines call and the data that is affected.

The graph for `INS_INPUT` (INS-D-3) shows five connected components. The packager graphs reveal that `INS_INPUT` makes calls to subprograms in three packages (`ins_fcc_io`, `ins_model_io`, and `ins_ad_io`) and the subprograms in any given connected component are declared in the same package. For example, subprograms `INS021`, `INS022`, `INS023`, `INS010`, and `INS108` are all declared in `ins_fcc_io`. The assertion is trivially true for the other subprograms in the `INS_INPUT` graph (INS-D-3). We observe that no data flows between packages under the control of `INS_INPUT`, because the components in the graph are not connected. The *input functions* of these three packages are *not* data coupled.

INS-D-4 shows that the *output functions* are data coupled. The packager shows that subprograms `INSENV`, `INSPLN`, `INSOTW`, and `INSAVL` are declared in `ins_model_io`. `INSADO` is declared in `ins_ad_io`. `INS501` and `INS502` are declared in `ins_fcc_io`. If we were to draw dashed lines to reflect these groupings, INS-D-4 would show the data coupling of the output functions among the packages.

The dataflow graphs do not plainly show the data coupling between the separate spans of control (i.e., input and output). For example, the graph for `INS_INPUT` (INS-D-3) shows that `PLNINS` writes variable `PSITQ`, and the graph for `INS_OUTPUT` (INS-D-4) shows that `INS501` reads (and writes) it. `PSITQ` is written by a subprogram in `ins_model_io` and read by one in `ins_fcc_io`. INS-P-5 clearly shows this data binding and others between the two packages.

This is another example of one way in which the Packager and DFD views are complementary. As we studied these graphs, we realized that grouping subprograms in a DFD graph by package may be another good way to simplify the DFD. The `INS_OUTPUT` DFD, for example, would be replaced by one graph with three of what we might call "package transform" nodes, and the three corresponding subgraphs. This technique merits further investigation.

We noticed that INS-P-5 indicates the wrong direction for the PSITQ data binding, and for others as well. In each case where the Packager view is wrong, the DFD is correct. We traced the problem to the use of "reduction over union" in the implementation of the pak-edge-global-exact-provisions and pak-edge-global-exact-requisitions attributes in the Packager. The attributes should be corrected in a production version of the RET to take into account the direction of the base edges.

2.4 MFD SUBSYSTEM

Packager — Exhibit MFD-P contains samples of the Packager view for the MFD subsystem. We organized the Packager graphs for MFD (MFD-P-1 through MFD-P-9) according to the call edges or relations. The call edges appear as thick arrows. This illustrates that the Packager graphs are effectively call diagrams when they are configured to show call edges instead of data bindings. We achieved a layered design among the packages and note that the interconnection strength numbers shown on MFD-P-1 are small relative to some of the other subsystems.

We will not describe each of the graphs due to the size of the MFD subsystem. We note that package mfd_keyboard (MFD-P-7) contains a number of subprograms that call each other, but the greatest number of data bindings that the package shares with another is four. We find this high cohesion and small coupling pleasing.

The graphs entitled mfd_d* (MFD-P-8) and mfd_dmf* (MFD-P-9) correspond with package nodes labeled mfd_dmf and mfd_d, respectively, in the library graph. We changed the labels while printing the hardcopy; the Packager does *not* automatically generate asterisks.

DFD — Exhibit MFD-D contains samples of the DFD view for the MFD subsystem. We will not give details on the MFD Dataflow Diagrams (MFD-D-1 through MFD-D-34) due to the size of the MFD subsystem. We note that the number of graphs is great in part because MFD makes nine calls to subprogram CHGREQ, resulting in nine graphs, and MFD_COMPUTE and STRSKB are printed on four pages each due to their size.

2.5 RLT SUBSYSTEM

Packager — Exhibit RLT-P contains samples of the Packager view for the RLT subsystem. RLT-P-1 shows the library units. The Packager generates upper-case node

names. We entered only lower-case node names to distinguish packages formed while interacting with the Packager from the generated nodes. In most of the graphs, the lower-case nodes represent *packages* and the upper-case nodes represent *subprograms*.

The Packager generates some of the *packages* in the library graph (RLT-P-1) and they have upper-case names, so the convention doesn't hold at the library level. It's not hard to distinguish packages from subprograms in the library graph in practice because the engineer sees the packages appear on the screen as they are generated. In retrospect, we should have used ellipses for packages. We didn't do that at first because we wanted to discriminate another way, such as with dashed rectangles, but found INTERVISTA didn't support them. We should have reverted to the ellipse, but never did.

The Packager generated the nodes without edges in RLT-P-1. All but two of them represent package specifications that contain only data object declarations. We refer to these as *data packages*. They are derived from FORTRAN common blocks. The other two, EXTERNALS and INTRINSICS, contain subprogram declarations and body stubs derived from FORTRAN external subprograms and intrinsic functions, respectively.

The packages `rlt_interface`, `rlt_in`, and `rlt_out` contain subprograms that are not referenced from within the RLT subsystem. The edges between these packages and the subprogram `RLT_COMPUTE` show the data bindings. The hardcopy doesn't show it, but when we direct the Packager to show subprogram call edges, there are none. From this we deduce that `RLT_COMPUTE` is called only from outside of the RLT subsystem.

The labels on the edges show the direction of dataflow, but we didn't design the Packager view to serve as a dataflow diagram. One advantage of the DFD over the Packager for inspecting dataflow is that a data object appears only once in a DFD, but may be referenced more than once in a Packager graph. For example, the variable `IRARDQ` appears on the labels of both edges of `rlt_out`. This is a convenient feature of the Packager view, but would be disadvantageous in a dataflow diagram.

RLT-P-2 shows the subprograms in package `rlt_interface`. We know from interacting with this view that there are no call edges, so all three subprograms are called from another subsystem. RLT-P-3 shows edges with interconnection strength values of one. This implies that they represent call relations, which we confirmed by displaying the call edges (not shown). RLT-P-4 is similar, but shows one data binding.

RLT-P-5 and RLT-P-6 show the subprograms generated to take the place of FORTRAN intrinsic functions and external subprograms, respectively. The node labels show the subprogram names, formal parameter types, and return types (for functions), all automatically generated by the RET prototype. The graph for the EXTERNALS package shows three subroutines and one function (IS_SET). The Packager generated all four external subprograms because we inadvertently omitted their FORTRAN source code files from the analysis.

DFD — Exhibit RLT-D contains samples of the DFD view for the RLT subsystem. RLT-D-1 shows seven transform nodes. We will focus on the nonterminal transform node RLT_INPUT first. It represents the transformation of one set of variables (DEGSEM, F066_FLAG, et al.) into another set (MWZ, PHIDQ, et al.). The graph shows this via arrows directed from the buffer with the first set towards RLT_INPUT, and arrows directed from RLT_INPUT towards the buffer with the second set. RLT-D-1 shows several transformations, of which RLT_INPUT is one, at the highest level of abstraction provided in the RLT DFD.

RLT-D-2 shows a more detailed level of abstraction for *the same transformation* that we discussed above. RLT-D-1 clumps the variables together as discussed, and summarizes the transformation of one set into the other. RLT-D-2 elaborates on how the individual variables in the sets are transformed. It shows that RLT_REF_IN transforms DEGSEM into PHIDQ, and statements in the body of RLT_INPUT transform F066_FLAG into MWZ.

RLT-D-1 suffers from an anomaly that causes too many arrows to local buffers in a graph. It shows that RLT_INPUT transforms the variables DBLINT, IF66J, DUMMY4, and PAD. These variables should *not* appear in the graph because they are local to the transforms by which they are referenced by and they are *not* used by another transform. As local variables that are not used by another transform, they are not relevant to the interprocedural dataflow. They are noise that should be removed from the DFD. Ignoring this anomaly, the variables transformed by RLT_INPUT in the first graph are the same ones that appear in the second graph. The second graph just shows more detail for this particular transformation.

RLT-D-3 suffers from an anomaly that causes too few arrows to local buffers. The identifier MUX_PACKET declares formal parameters in subprograms RLT_RL00n_OUT, where n is 1, 3, and 4, and a local variable in RLT_OUTPUT. RLT-D-3 shows that RLT_RL001_OUT transforms the buffer MUX_PACKET. The *buffer represents the local*

variable that is passed through a formal parameter that happens to have the same name. The buffer is thus not local to RLT_RL001_OUT, so the graph does show RLT_RL001_OUT transforming the buffer. This part is correct.

RLT-D-3 should also show an arrow from RLT_OUTPUT to MUX_PACKET. The MUX_PACKET buffer is local to RLT_OUTPUT, but because it is used by RLT_RL001_OUT, the graph should show that RLT_OUTPUT transforms the buffer. In contrast with the anomaly discussed above, this relationship is relevant to the interprocedural dataflow because the buffer is used by another transform. Both anomalies should be corrected in a production version of the RET.

All of the other buffers in RLT-D-3 are involved in the RLT_OUTPUT transformation on RLT-D-1. RLT-D-3 just shows more detail on the RLT_OUTPUT transformation than RLT-D-1. For example, RLT-D-1 shows that RLT_OUTPUT transforms RL001_FLAG, CURRENT_POWER, IRARDQ, and MRL01J into RL001 and RLD00p, where p is between 1 and 4. RLT-D-3 reveals that RLT_RLTADO takes IRARDQ and MRL01J into RLD00p, RLT_RL001_OUT derives RL001 from a different source, and RL001_FLAG and CURRENT_POWER are transformed by statements in the body of RLT_OUTPUT. Due to the missing arrow, we can't tell that RLT_OUTPUT feeds MUX_PACKET.

2.6 UFC SUBSYSTEM

Packager — Exhibit UFC-P contains samples of the Packager view for the UFC subsystem. We printed only a sample of the UFC subsystem because it is very large. The Packager graphs (UFC-P-1 through UFC-P-8) show the same kind of design, layered with little coupling, that is enjoyed by the MFD subsystem. The Packager graphs that we show are essentially call graphs. The Packager can display combinations of data binding and call edges, but this hardcopy shows only call edges.

DFD — Exhibit UFC-D contains samples of the DFD view for the UFC subsystem. The top-level DFD (UFC-D-1) shows the same characteristic structure as the other subsystems. It includes initialization, termination, suspension, input, output, and compute routines. We do not give details for the UFC subsystem shown in UFC-D-1 through UFC-D-21, but note one important point.

The engineer can make use of information in even the most complicated views by taking advantage of the filtering capabilities of the RET prototype. UFC-D-9, for example,

is very complicated and the hardcopy version does not appear to be very useful. Nevertheless, the engineer can answer specific questions by hiding some of the nodes or edges. The engineer may, for example, display only call edges from subprogram UFCBIT.

Another feature that helps the engineer manage complexity is the automatic highlighting of adjacent nodes in the Packager and DFD views, which can't be shown in the hardcopy. The RET prototype outlines all nodes that are adjacent to (have an edge to) the node under the cursor. The outlines appear and disappear automatically as the engineer moves the cursor. The effect is that in a complicated view, the engineer can visually scan for the nodes that, say, are called by, or share a data binding with a particular node.

2.7 SMS SUBSYSTEM

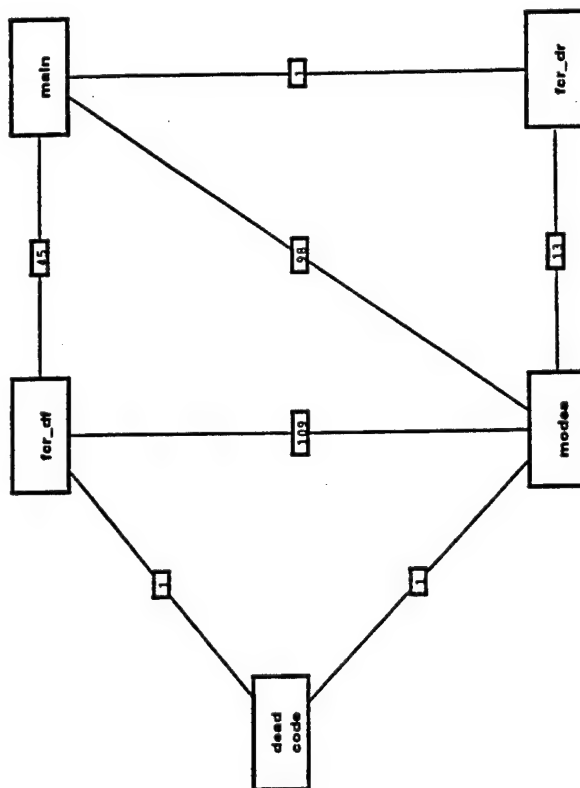
We did not analyze the SMS subsystem.

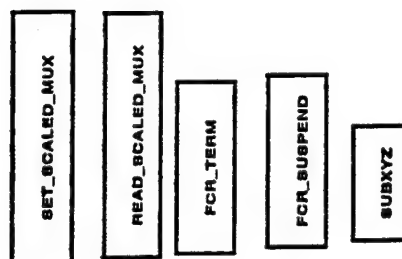
REFERENCES

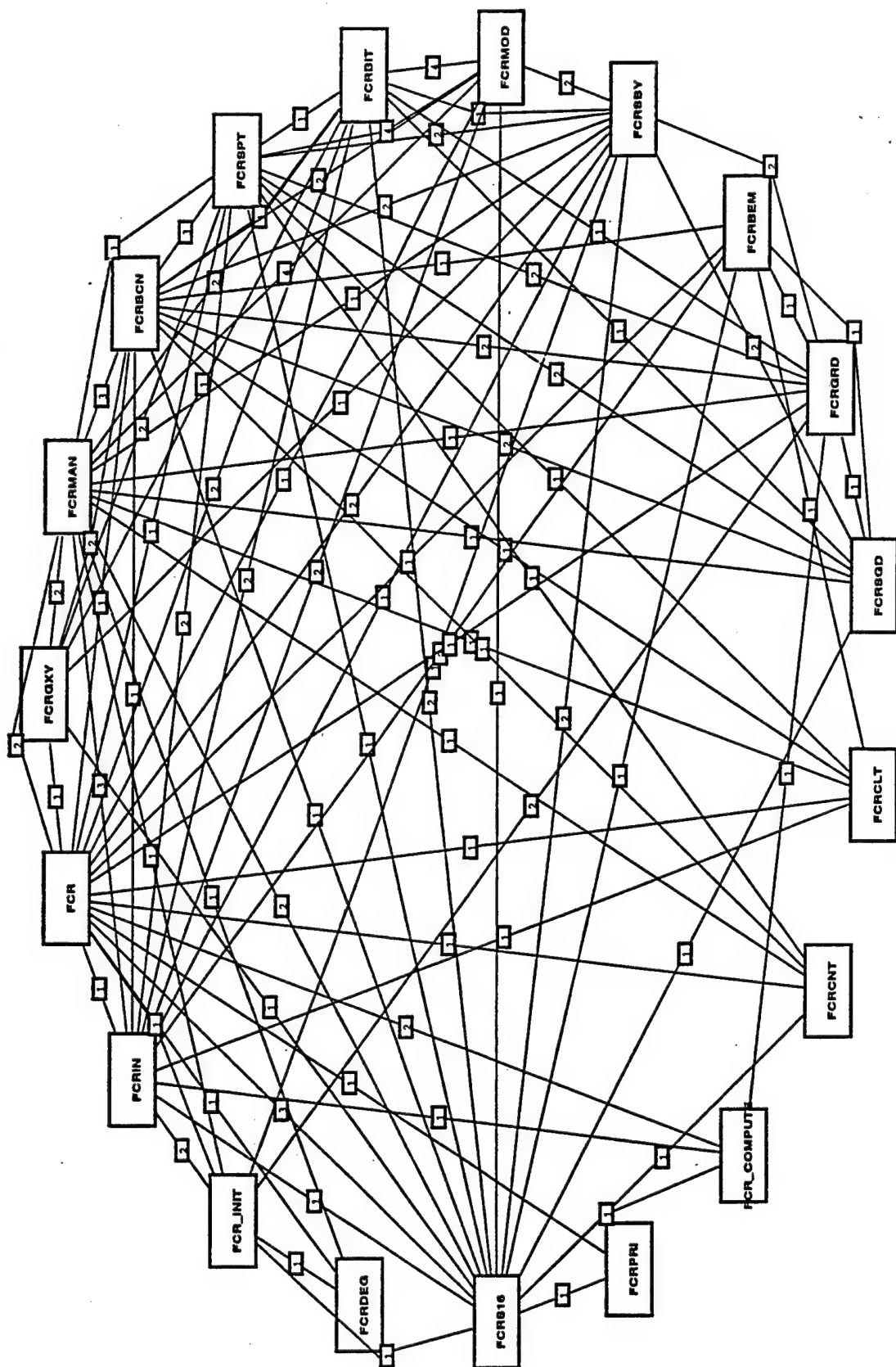
1. D.E. Wilkening, Avionics Software Reengineering Technology (ASRET) Project Final Report, Volume I, Project Summary, Account, and Results, TASC Technical Report TR-6661-4, TASC, Inc., Reading, Massachusetts, 5 May 1995.
2. Corbi, T.A., Program Understanding: Challenge for the 1990s, IBM Systems Journal 28(2), 294-306 (1989).
3. Wilkening, D.E., Kreutzfeld, R.J., and Loyall, J.P., Avionics Software Reengineering Technology (ASRET) Software Reengineering Study Report, Technical Report TR-6661-1, TASC, Reading, Massachusetts, 17 February 1993.
4. D.E., Wilkening, J.P. Loyall, Software Requirements Specification for the Avionics Software Reengineering Tool (RET) Prototype System, RET-SRS-01. TASC Technical Report TR-6661-2. TASC, Reading, Massachusetts, May 1993.
5. D.E. Wilkening, J.P. Loyall, Software Design Document for the Avionics Software Reengineering tool (RET) Prototype System, RET-SDD-01. TASC Technical Report TR-6661-3. TASC, Reading, Massachusetts, August 1993.

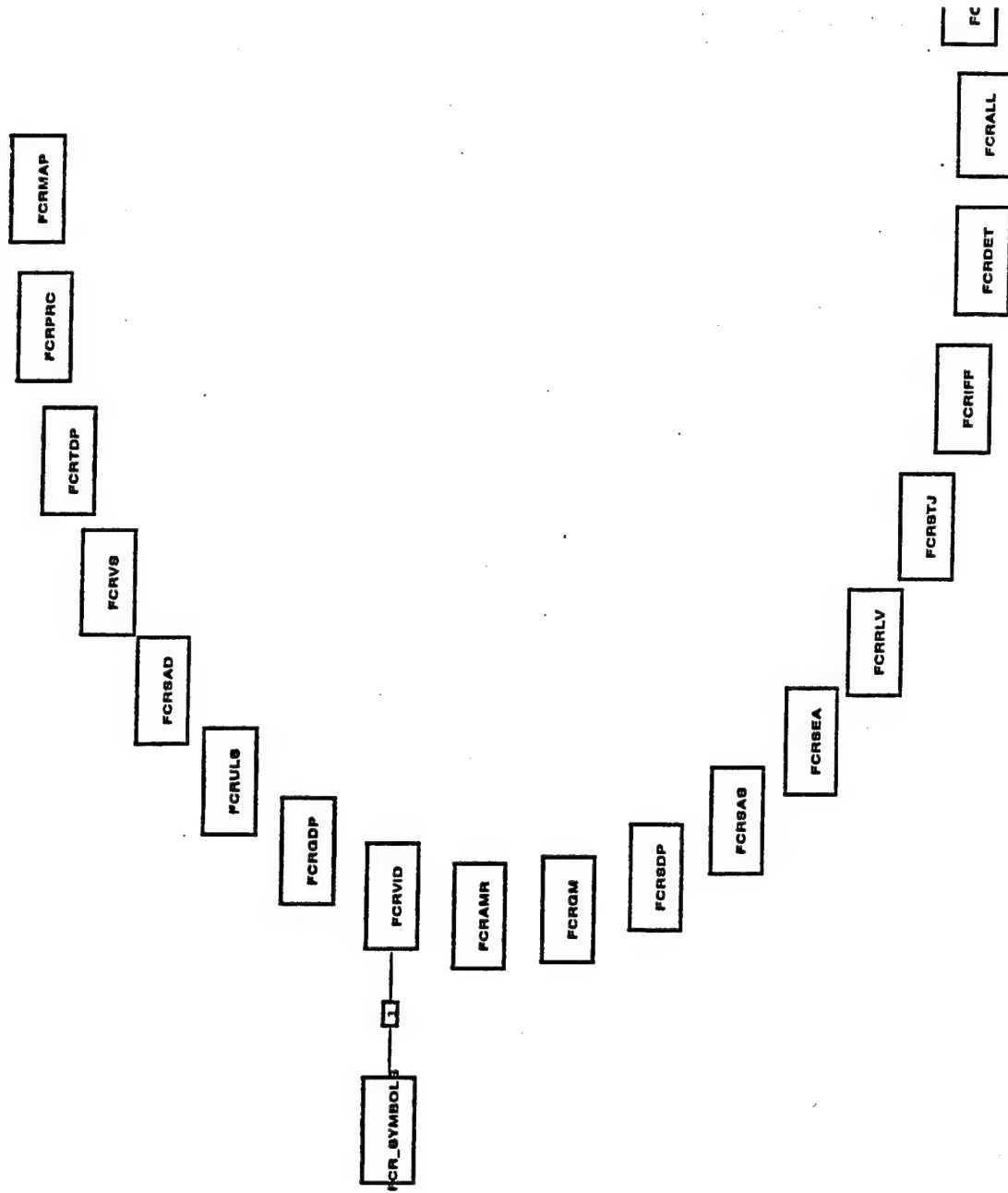
APPENDIX A

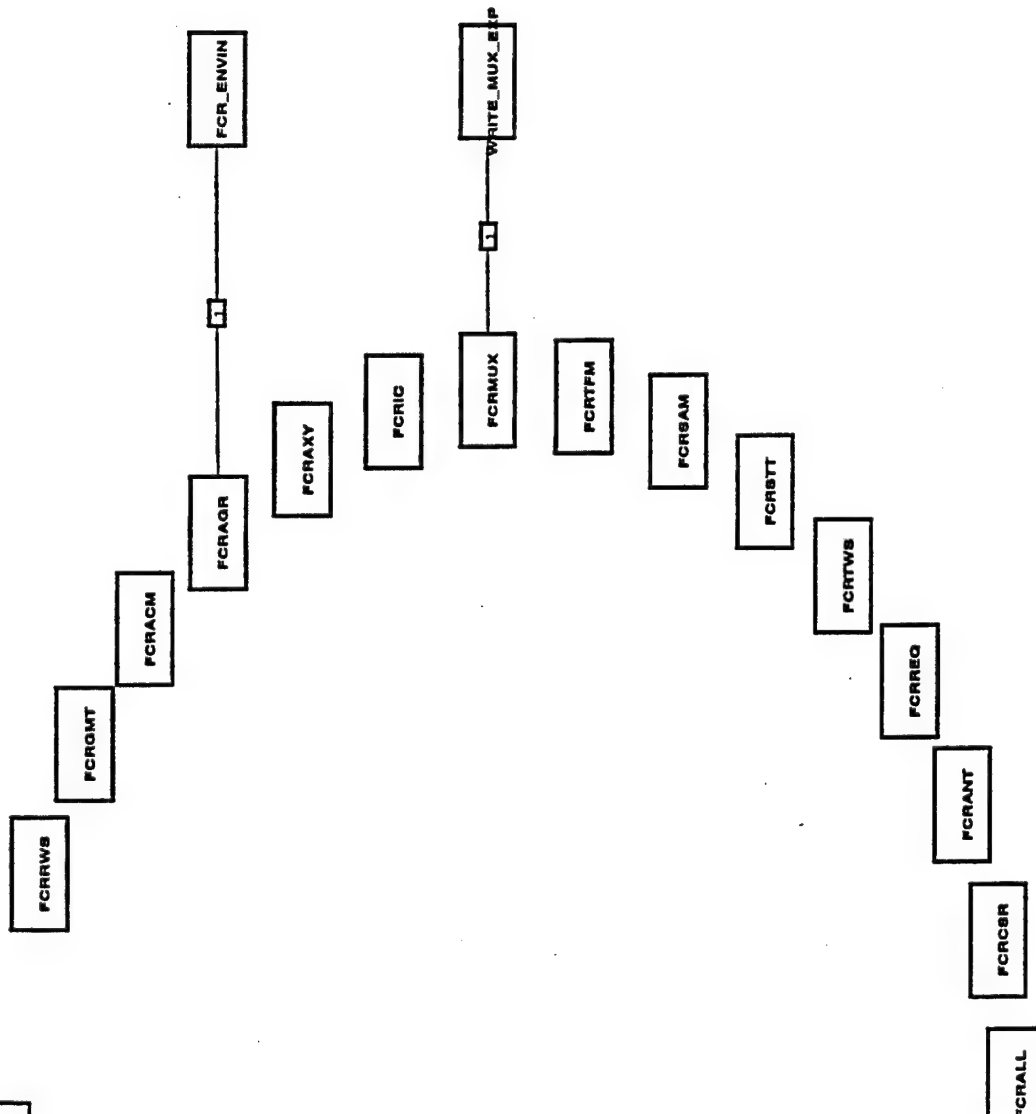
EXHIBIT FCR-P FCR Subsystem Packager Views

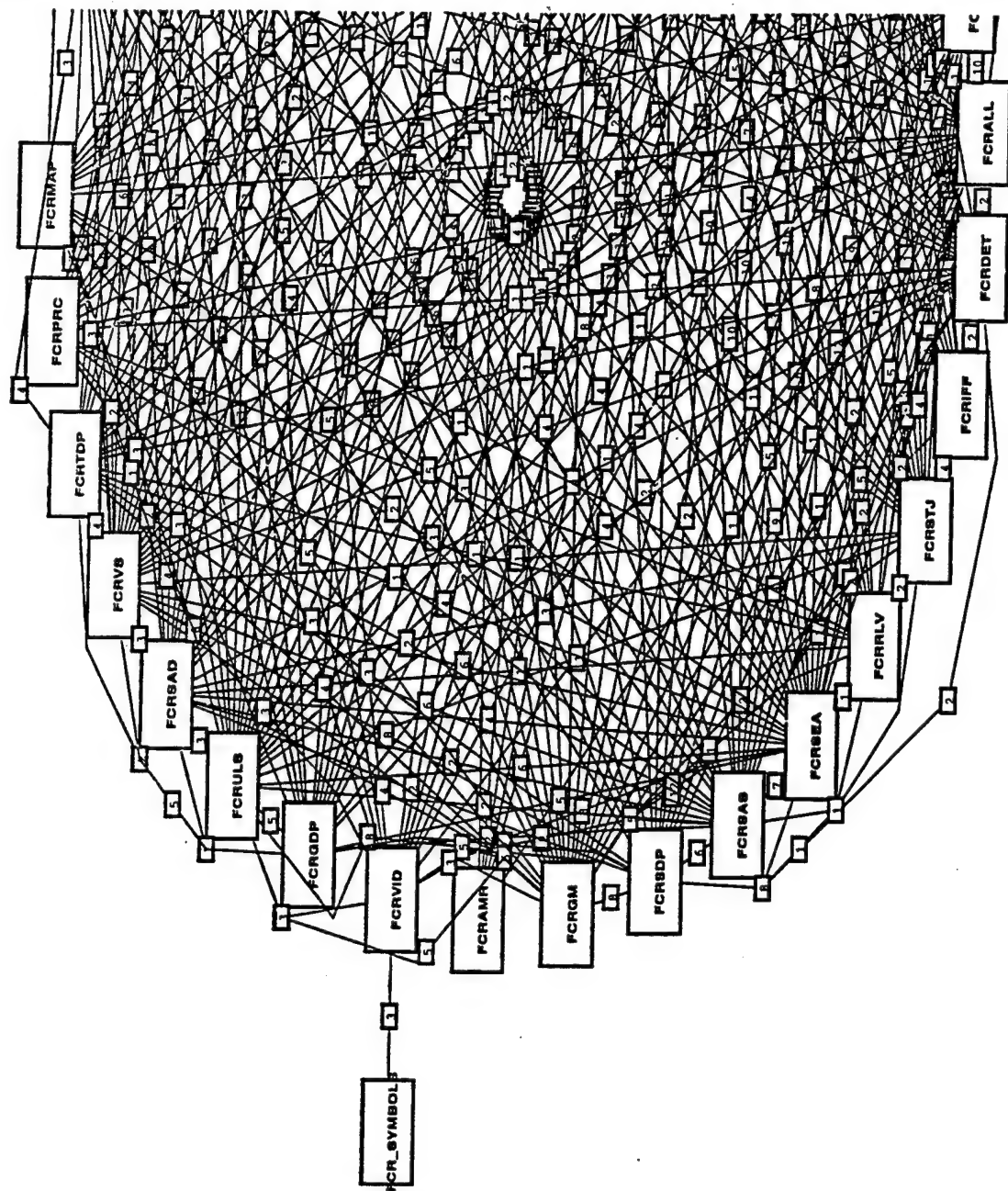


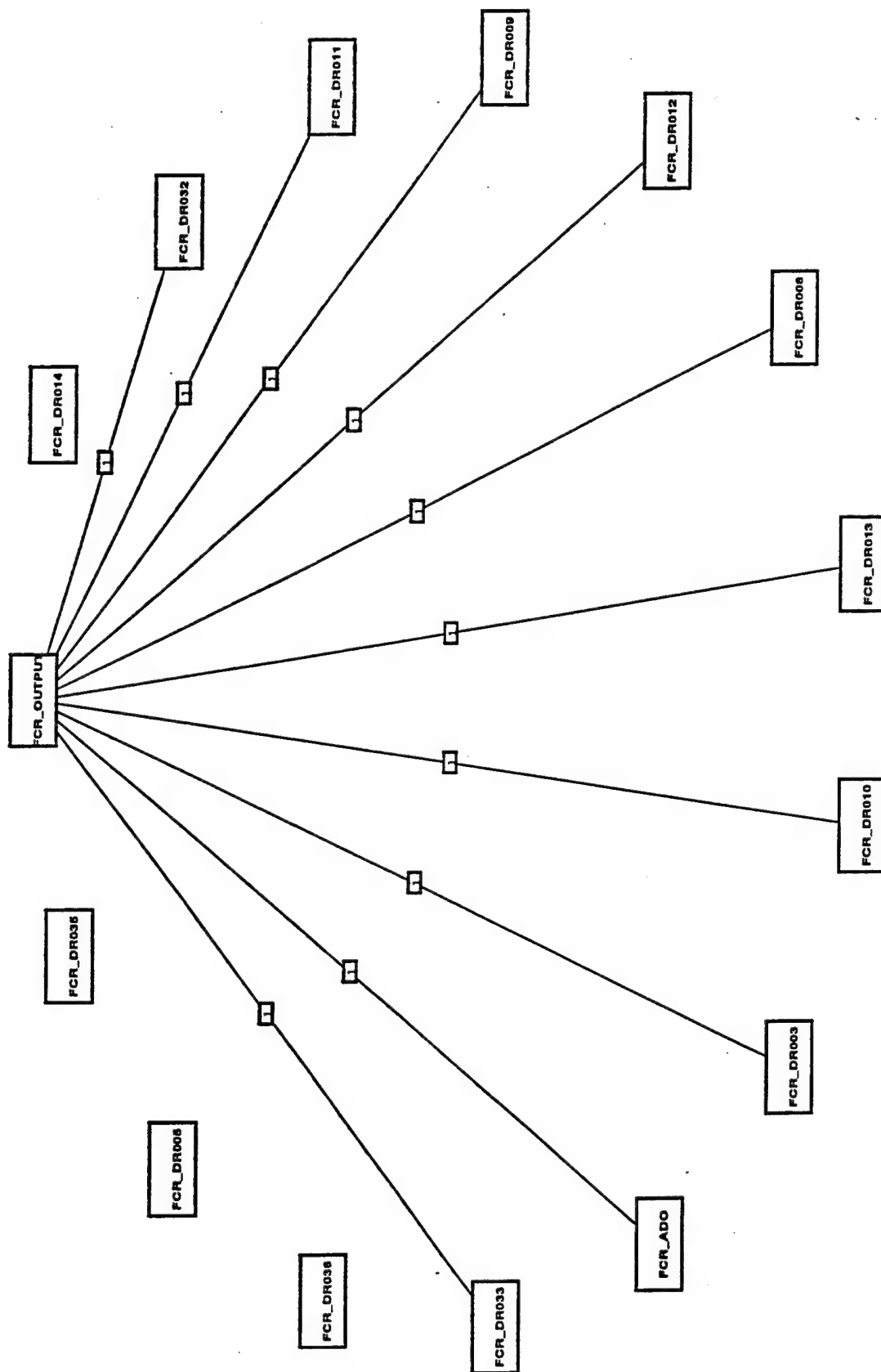






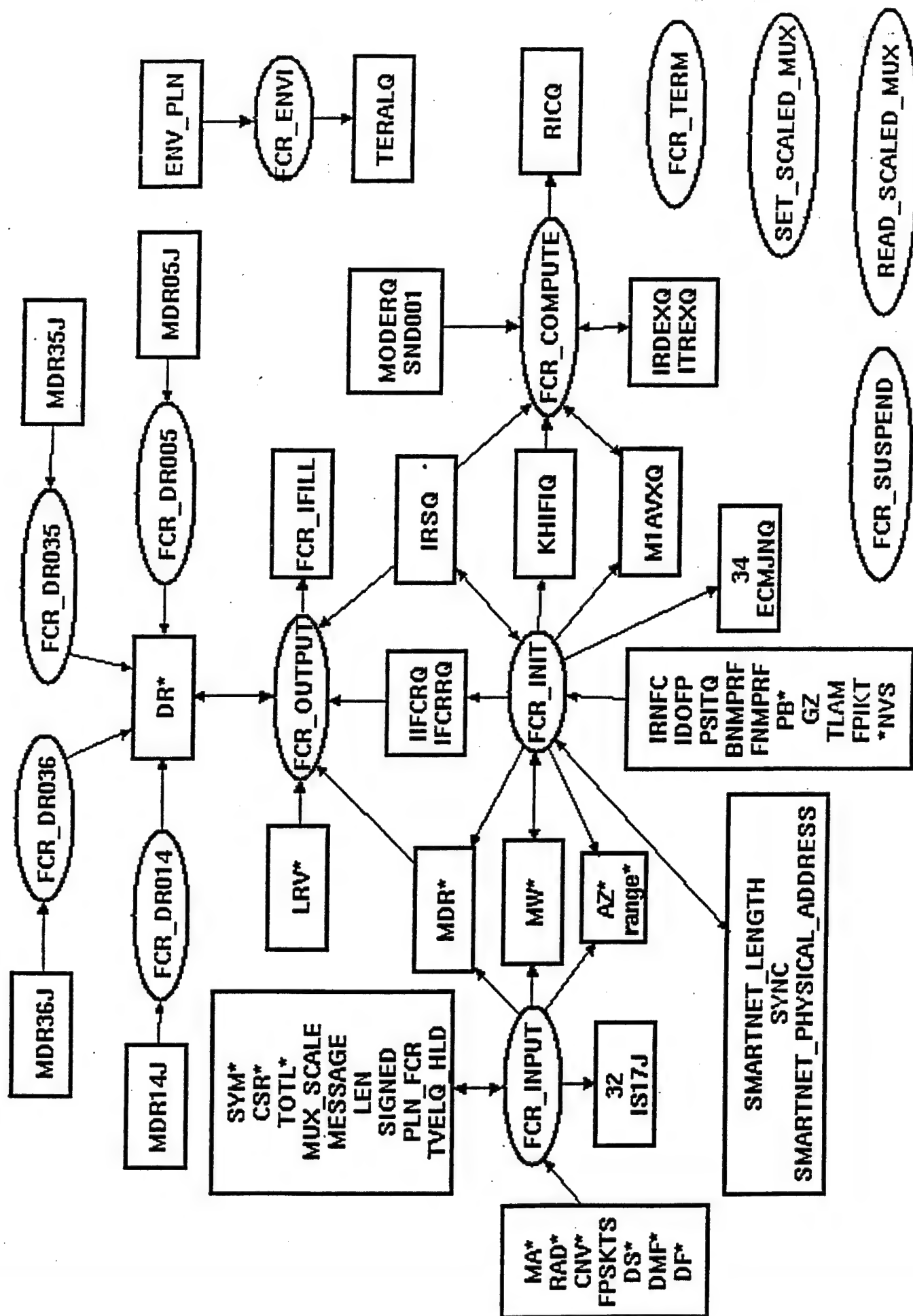


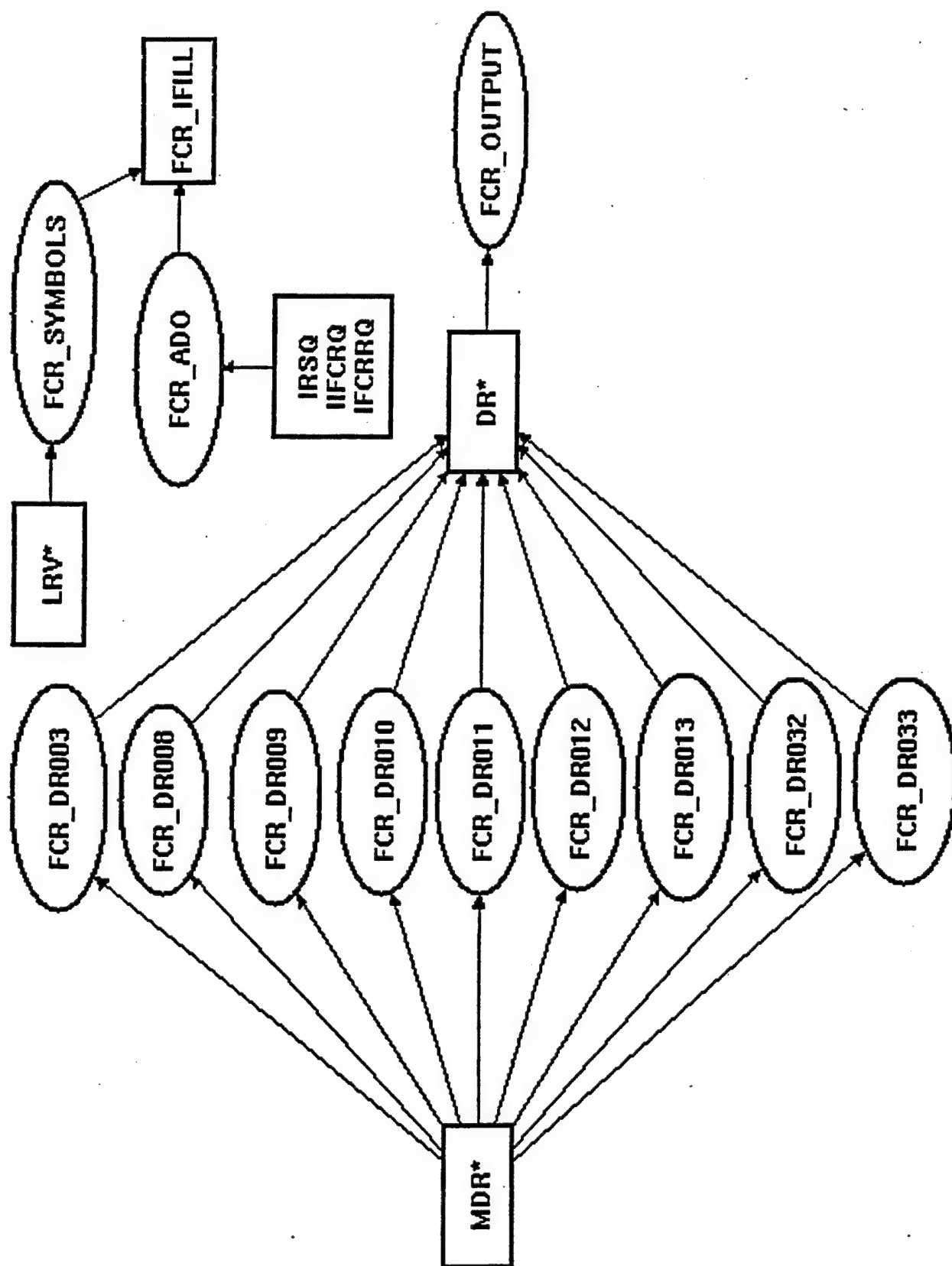


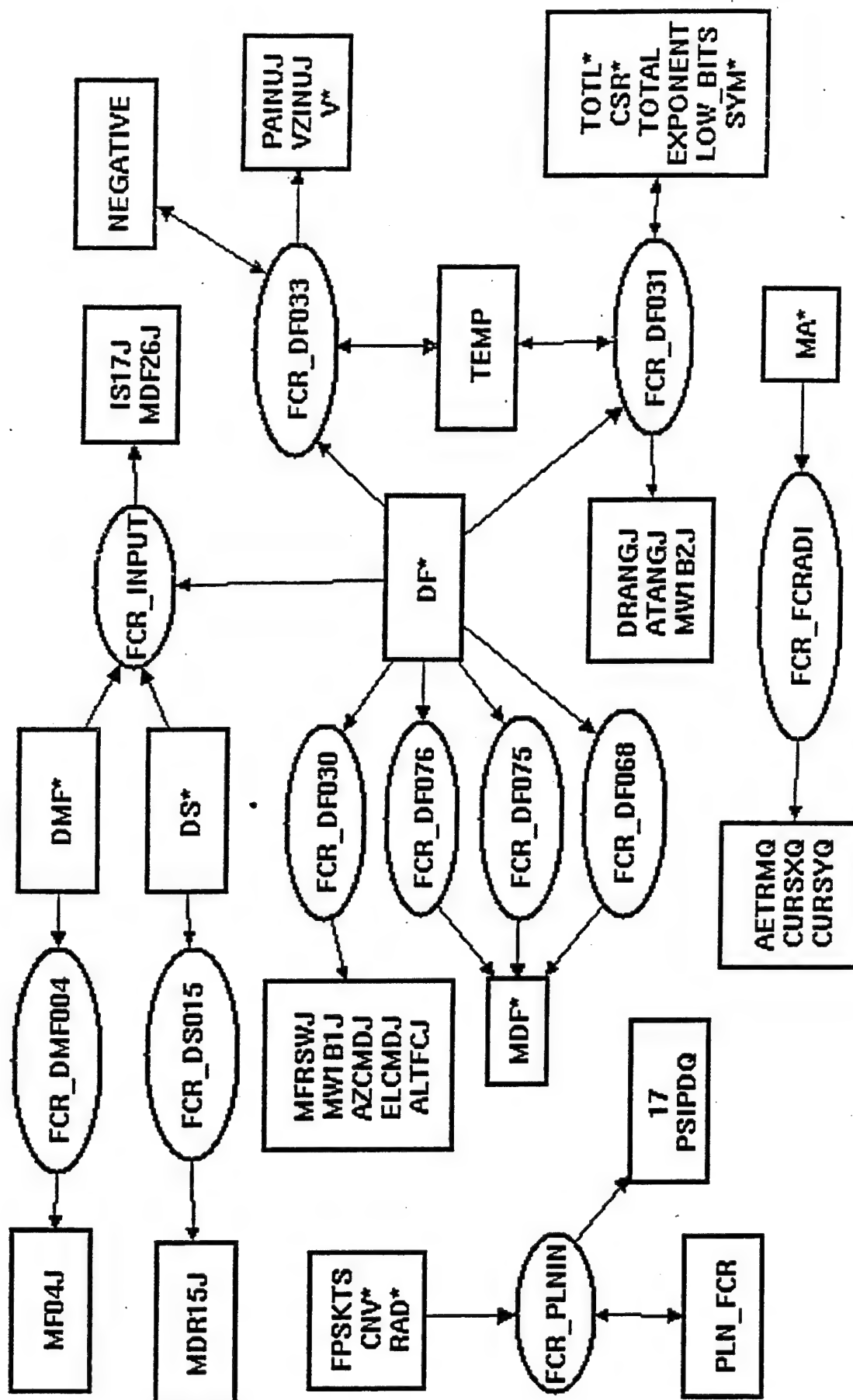


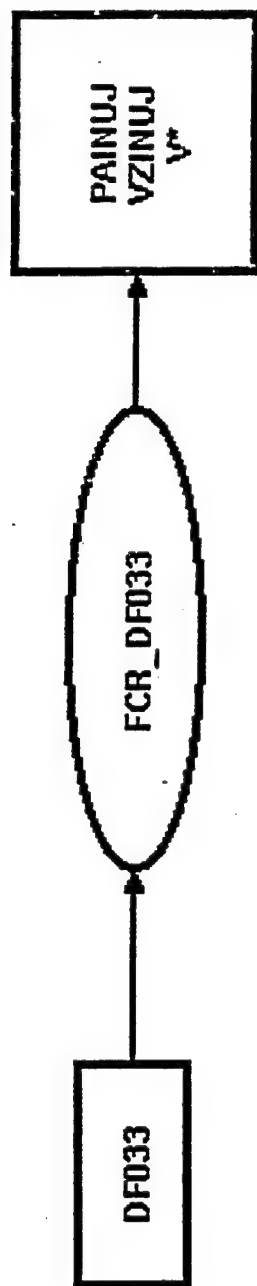
APPENDIX B

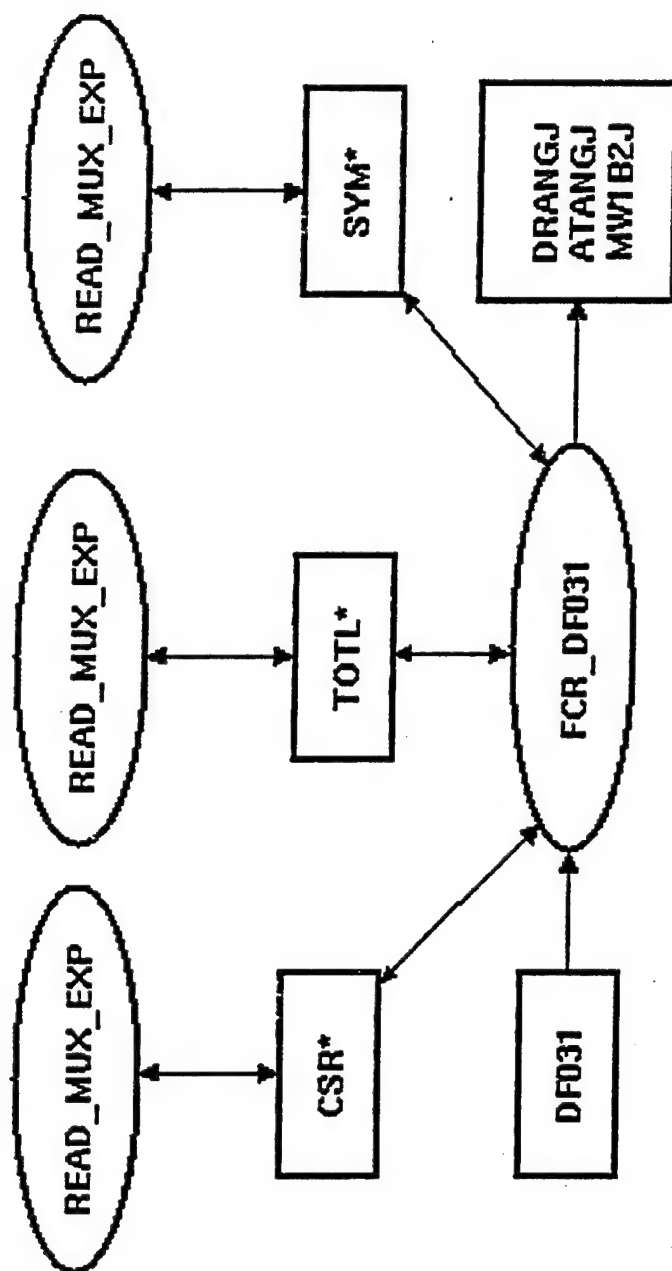
EXHIBIT FCR-D FCR Subsystem DFD Views



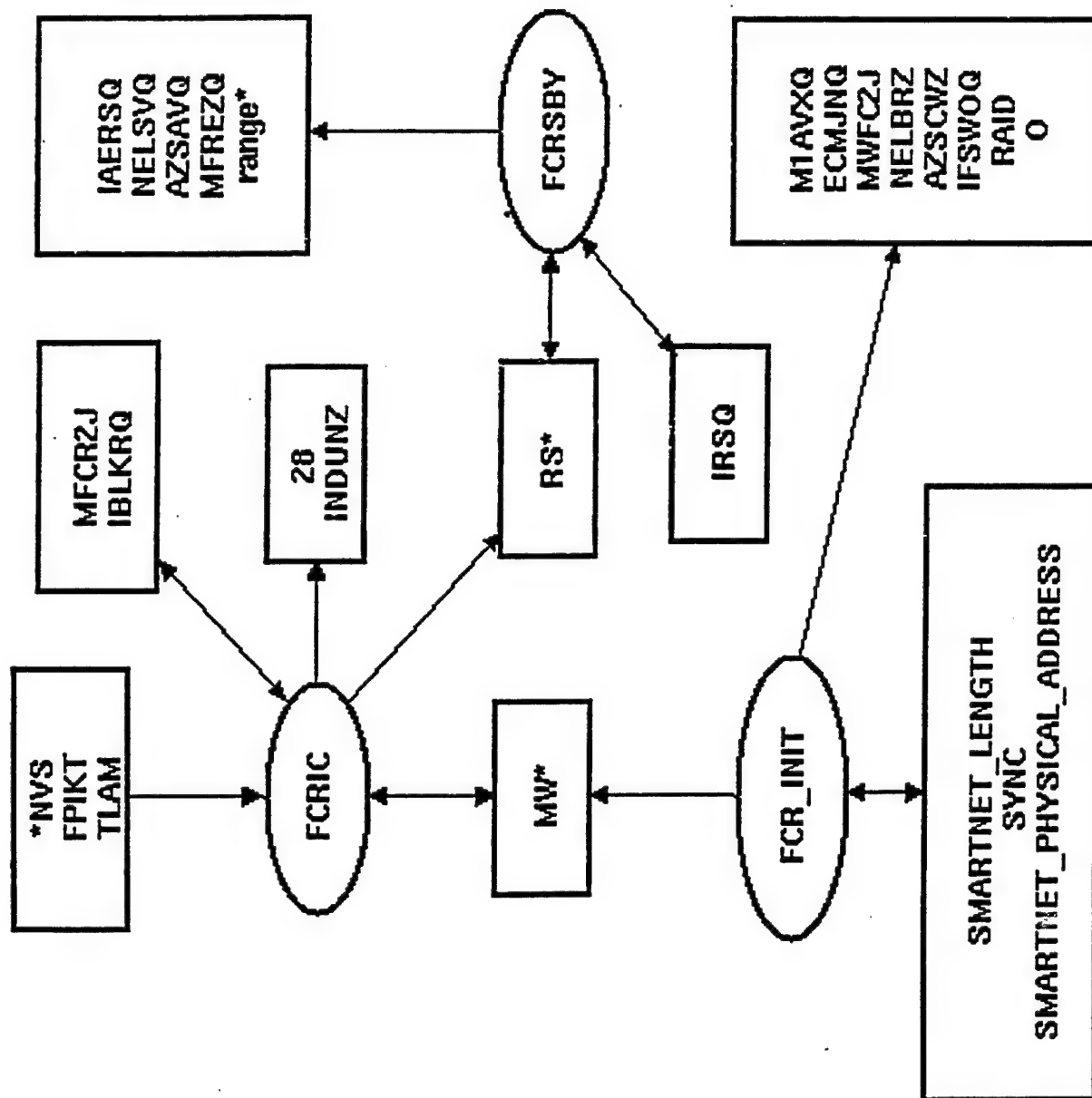




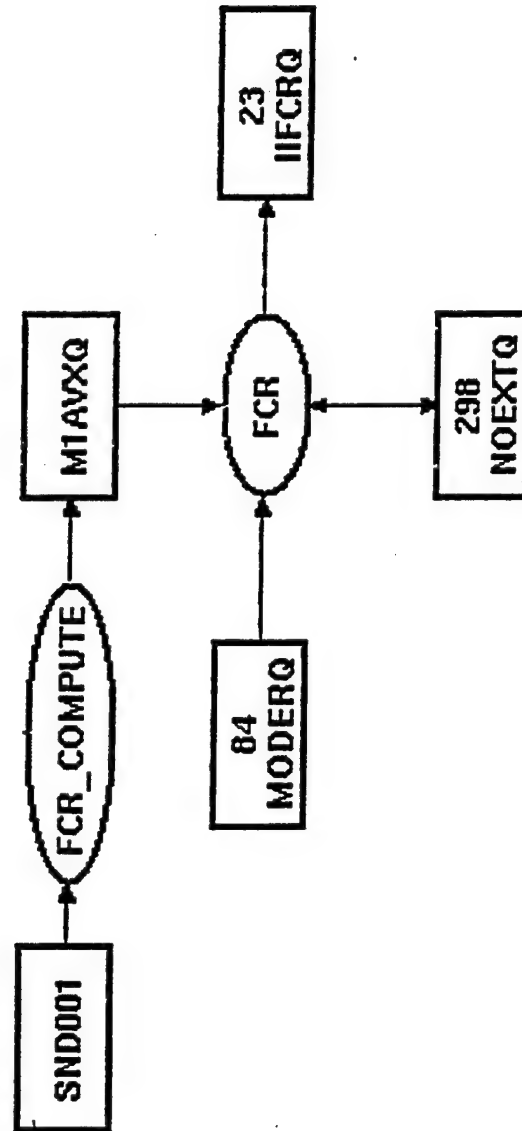




FCR_INIT



FCR_COMPUTE



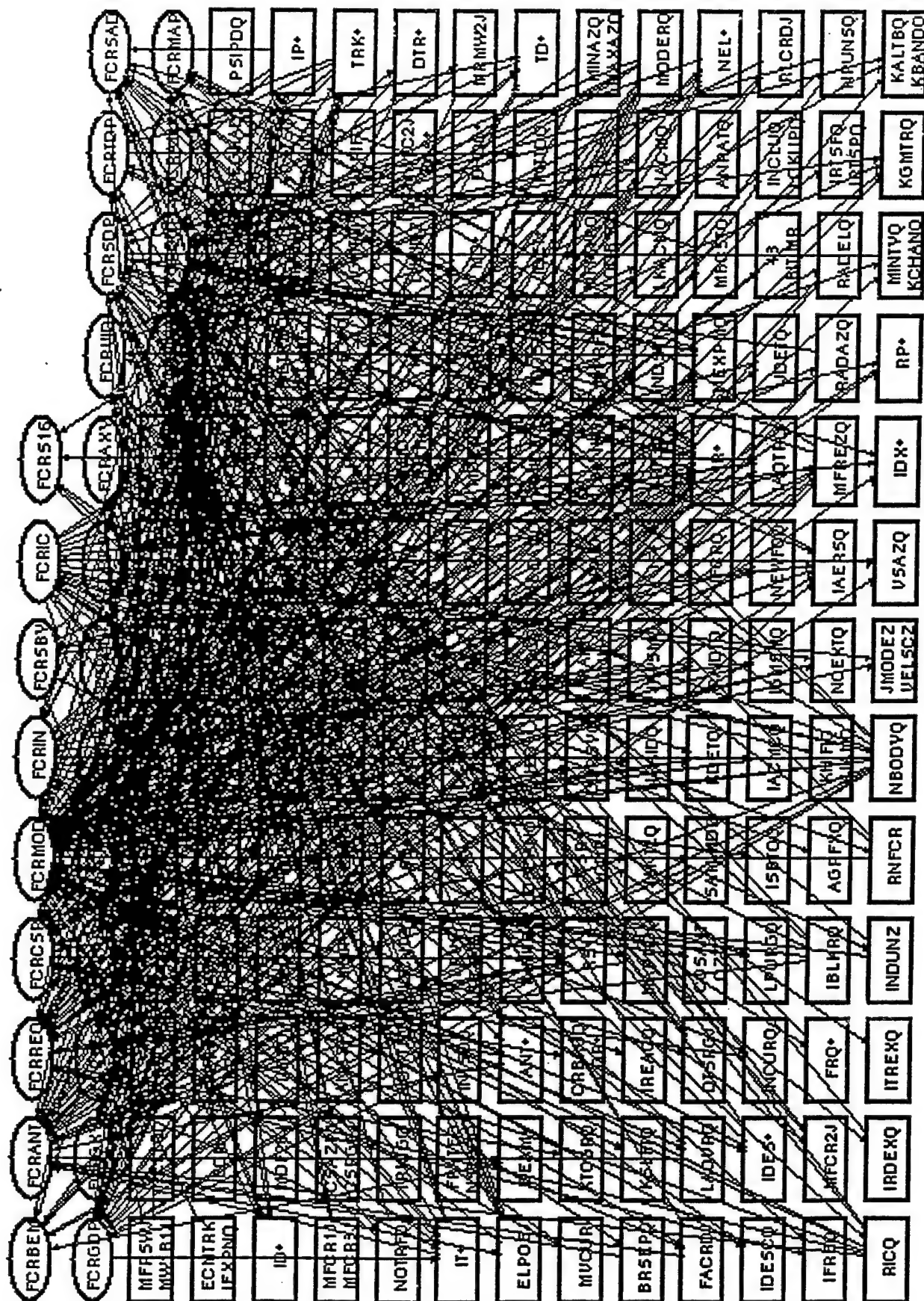
FCR

[illegible]

[illegible]

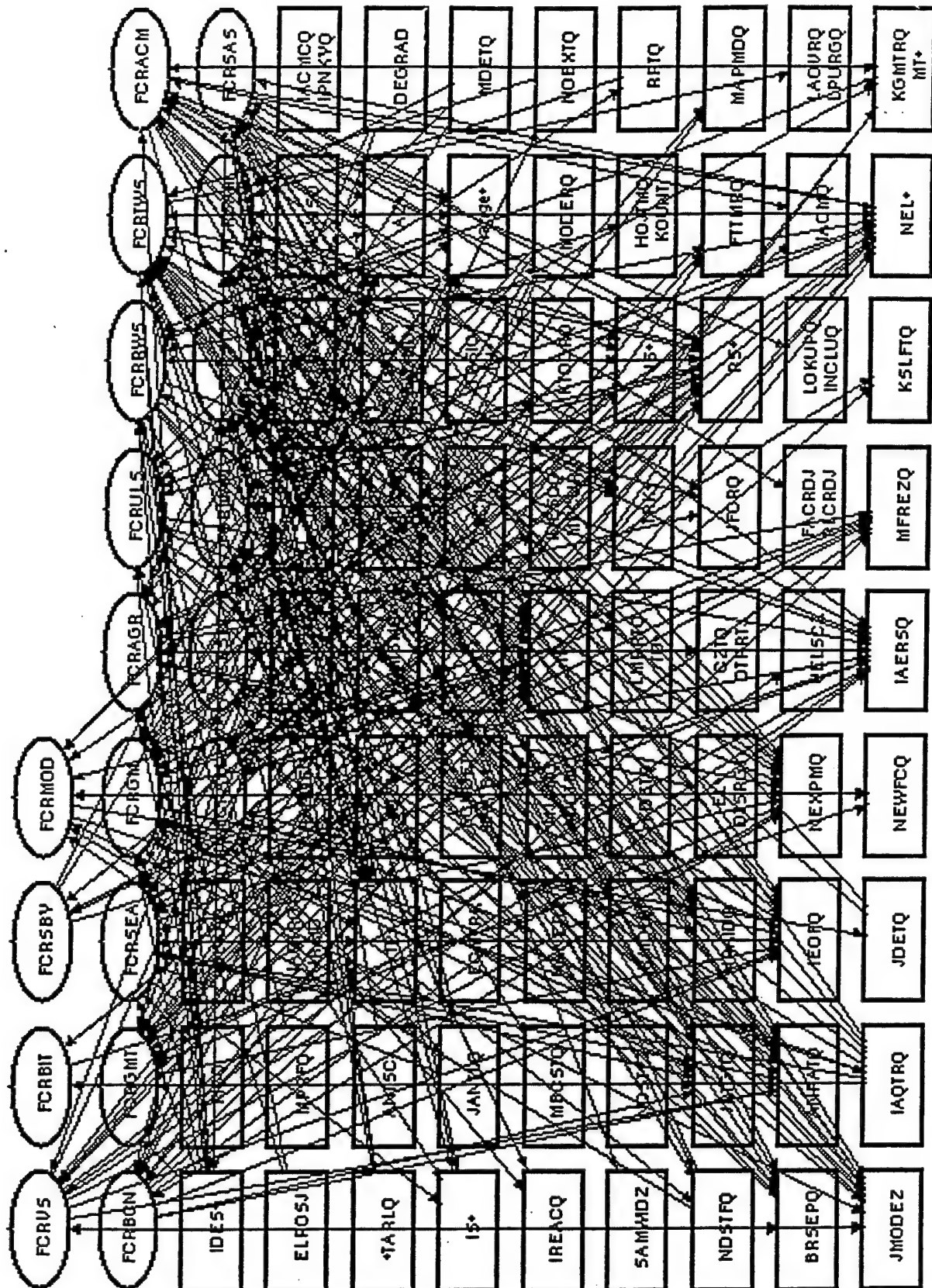
F CRS16

FCRBDI	FCRANT	FCRREQ	FCRCSB	FCRMOD	FCRIN	FCR5BV	FCRIC	FCR51G	FCRVID	FCR5DP	FCR1DP	FCRSAD
FCRGDB	FCRGXV	FCRSGD	FCRPRU	FCRFTM	FCR1FF	FCRCNT	FCRPRO	FCRAXV	FCRMUX	FCROUT	FCRAMR	FCRMAP
MFR5WJ MW1R1J	TELLQ	TBRGQ	range*	TU*	*TARLQ	*NUS FBIKT	PSIIQ	M1AUXQ	U*	PAINUJ	DEGRAD	PSIPDQ
ECMTRK JEXPNQ	ECM*	IPCKOX BBTQ	JAM*	CBQ	RADDEG	ELCMDJ	AZ*	LT*	AETRMQ ATANGJ	MIF04J	AUSTSQ DRANGJ	IP*
ID*	MDF26J	TIMEQ	CUR*	MW*	SYN*	ECMDET ALTECJ	MITYPQ	KTUBQ	NBFCRQ J517J	IGMITQ	IPRIFQ	TRK*
MFCR1J MFCR3J	CSAZPJ C5B0PJ	ANAZPJ	MDR*	TEMP	MANTISS MESSAG	LTG5ZQ	TACZQ TACYO	IS*	MT*	LNKIDZ	MWFC2J DTA*	DTR*
NOTRFQ	TRKTGQ	LOWTFQ	IGCNTZ	SD*	GDIS*	LNGT*	NT*	MISIDZ MPLOYZ	LPRIQ	MD*	DHDG*	MRMW2J
IT*	FRATEQ NSTGDD	INTSDQ	LRU*	IA*	DEITMQ RG1ARQ	EL*	JA*	MDETQ	IDCSTQ	IDET*	INTTDQ	TD*
ELPOSJ	BEAM*	ANT*	IXMITQ	DSCANQ	IEO*	IBARCQ	NBEAMQ	KHIDEQ	MPREQ	MFPWRQ LDDR5Q	IC*	MINA2Q MAXA2Q
MUCUR	KTOGRQ	CRBRGQ COR*	RS*	CSR*	CURS5Q	RAID GNM*	IRAID	JBLNKQ	NDSTFQ	LKACMQ	IACMQ	MODERQ
BRSEPO	KSLFTQ	IREACQ	MAPMDQ	JANTLQ	NRAIDQ	JTW5MQ	J5*	JRTGTQ	NEXPMQ	MBCSTQ	ANRATQ	NEL*
FACRDJ	LAOURQ	DTSRGJ	CO SANT CZTQ	SAMWIDZ	KDETQ	RAIDTQ	JFCRQ	IR*	JAQTRQ	43 BITIMR	INCLUQ LOKUPQ	RLCRDJ
IDESQ	IDES*	INCURQ	LPURGQ	ISBTOQ	IACMCQ	IGM5MQ	NEWFCQ	IAQTRQ	RADA2Q	RADELQ	IRTSFQ JRT5PQ	NRUN5Q
IFREQ	MFCR2J	FRQ*	IBLKRQ	AGRFQ	KHFRQ KJAMQ	NOEXTQ	IAERSQ	MFREZQ	RP*	MINYQ KCHANO	KGMITQ	KALTBQ KRANDQ
RICQ	IRDEXQ	ITREXQ	INDUNZ	RNFCR	NBODYQ	JMODEZ JELSCZ	USA2Q	IDX*				

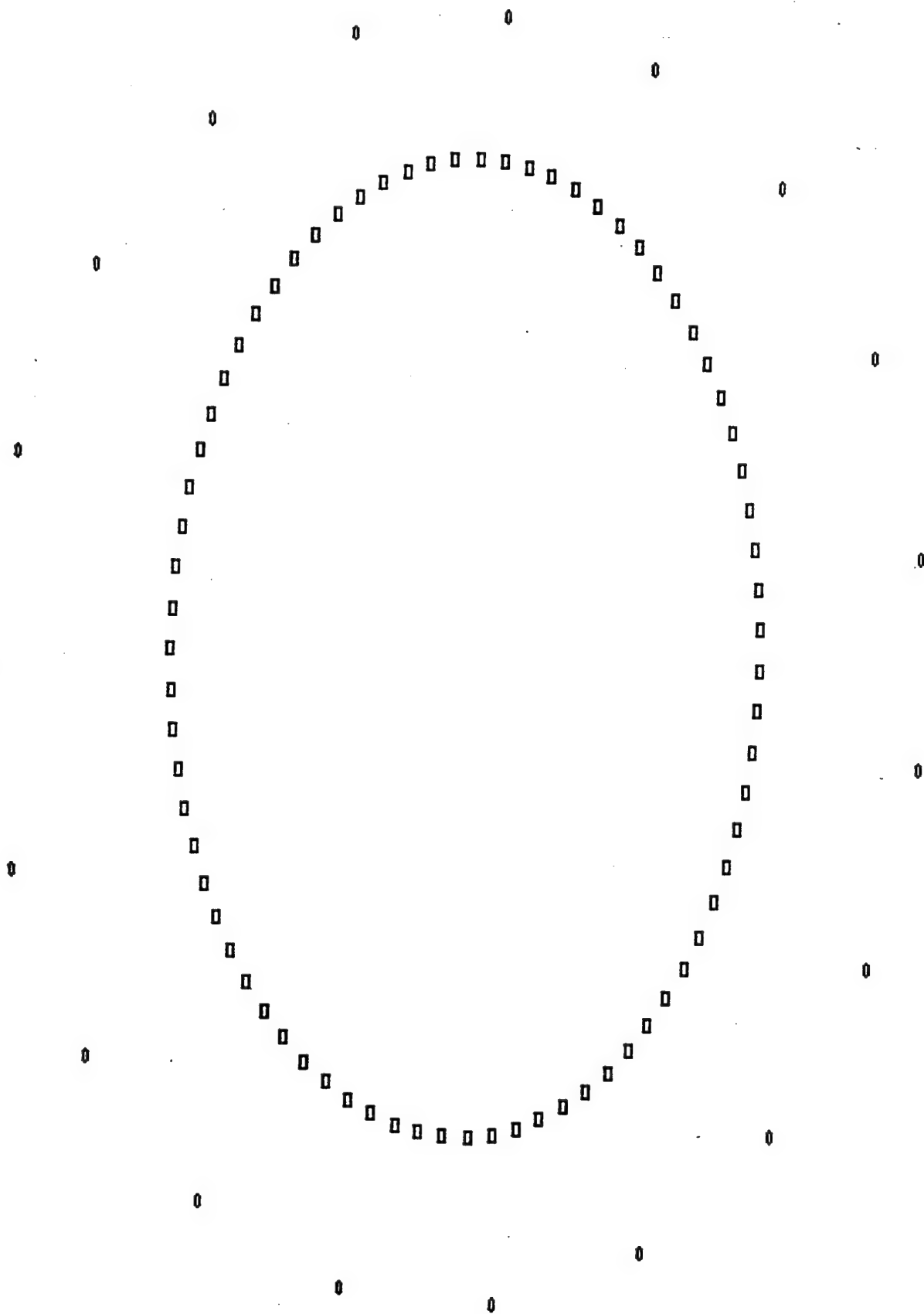


FCRMOD

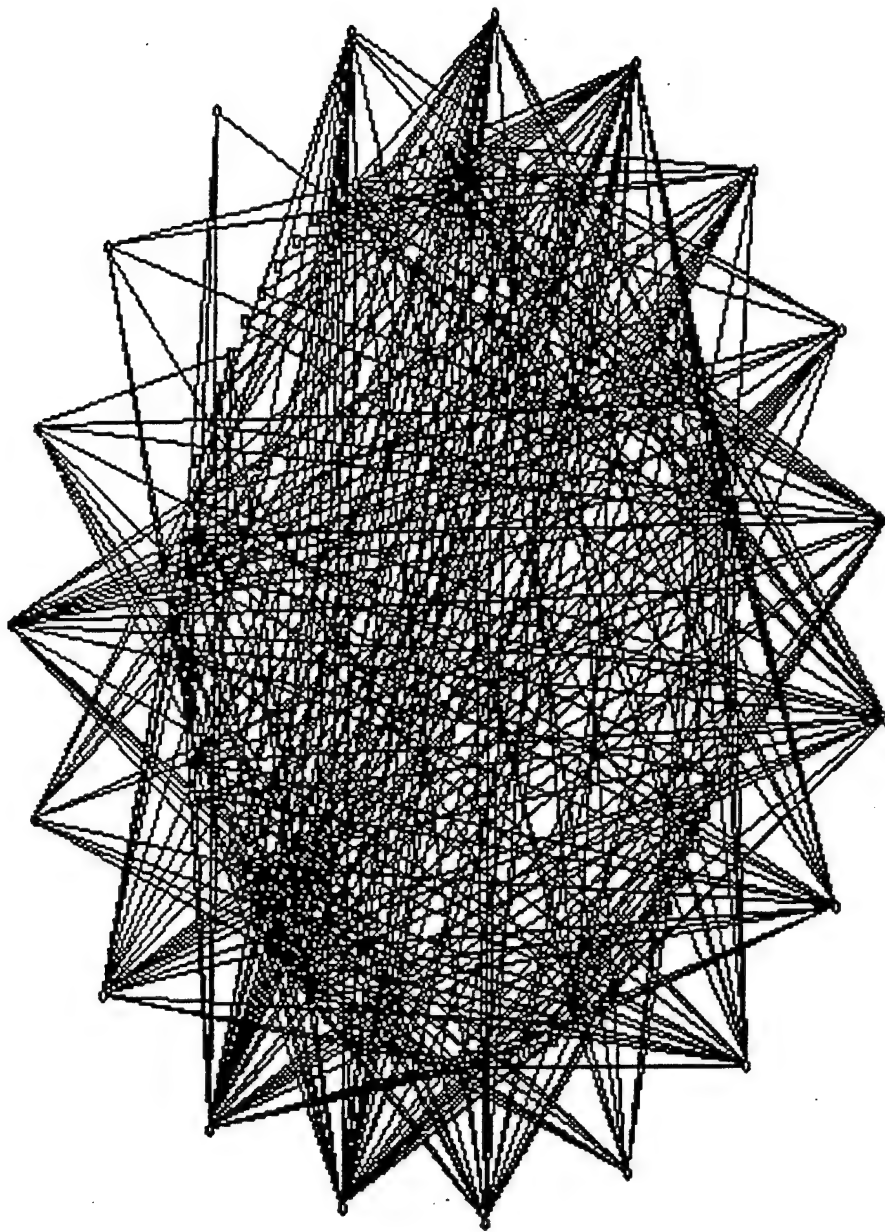
FCRUS	FCRBIT	FCRSBY	FCRMOD	FCRAGR	FCRULS	FCRRWS	FCRTWS	FCRACM
FCRBCN	FCRGM	FCRSEA	FCRGM	FCRSPT	FCRMAN	FCRSIT	FCRSAM	FCRSAS
IDES+	RICQ	IBLKRC	FCRALL	U+	LT+	NT+	IAVISQ	IACMCQ IPNKYQ
ELPOSJ	MPRFQ	JAQTRQ IRAID	RADDEG	IGMSMQ	RNFCR	AGRFXQ ELCMDJ	AZ+	DEGRAD
*TARLQ	ANTSCJ	IP+	IDET+	IT+	ECM+	TPSIQ TU+	range+	MDETQ
IS+	JANTLQ	ECMTRK	ECMDET MW1B1J	IR+	IDESCQ MVCUR	KTOGRQ	MODERQ	NOEXTQ
IREACQ	MBCSTQ	MANERZ	MJAMQ ACQTMQ	LMPRIQ TD+	TRK+	J5+	HOJTMQ KOUNT	RRTQ
SAMWIDZ	NDSTFZ	RAIDTQ	KDEIQ	CZIQ DTRRJ	JFCRQ	RS+	FTTMRQ	MAPMDQ
NDSTFQ	JRTGTQ	NRAIDQ	EL+ DTSRGJ	UELSCZ	FACRDJ RLCRDJ	LOKUPQ INCLUQ	IACMQ	LAQURQ LPURQ
BRSEPQ	ANRATQ	IEOFQ	NEXPMQ	IAERSQ	MFREZQ	KSLFTQ	NEL+	KGMTQ MT+
JMODEZ	IAQTRQ	JDEIQ	NEWFCQ					



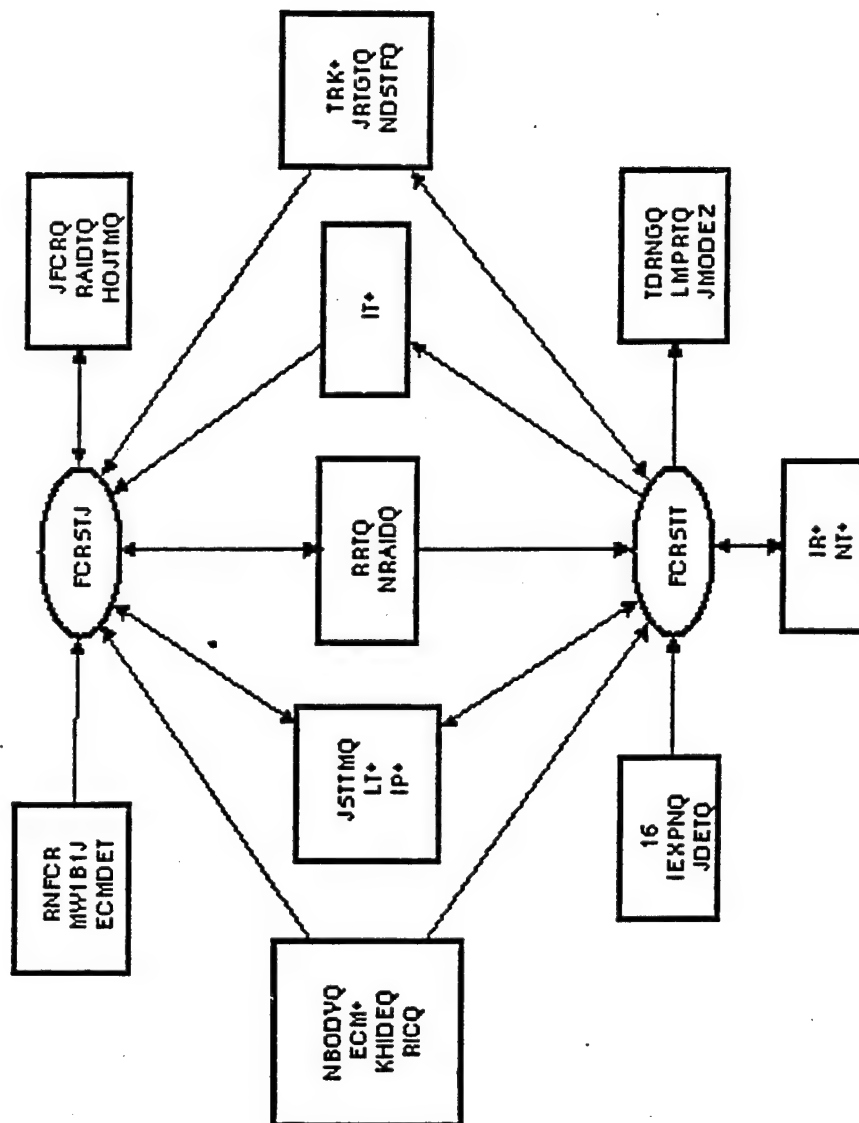
FCRMOD

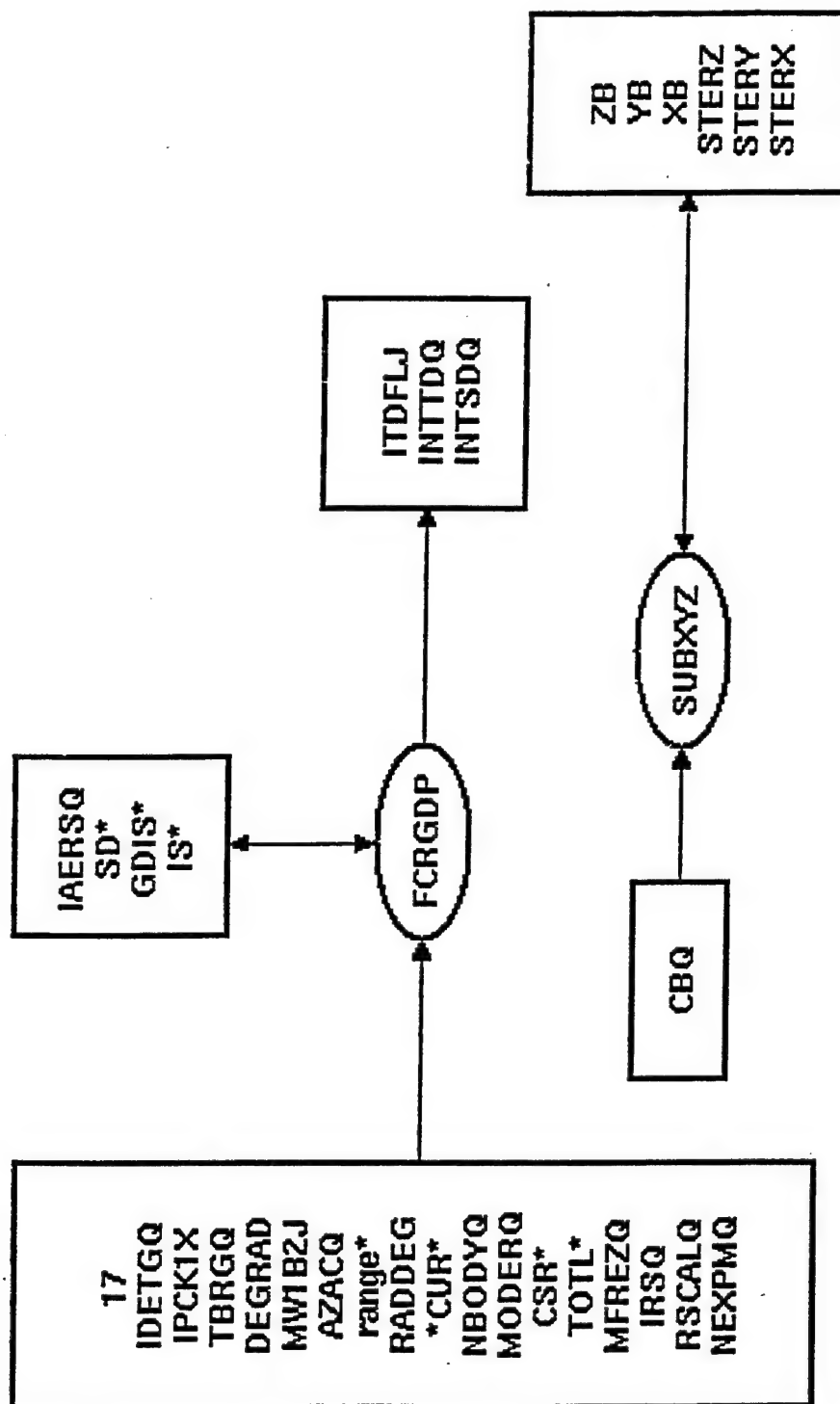


FCRMOD

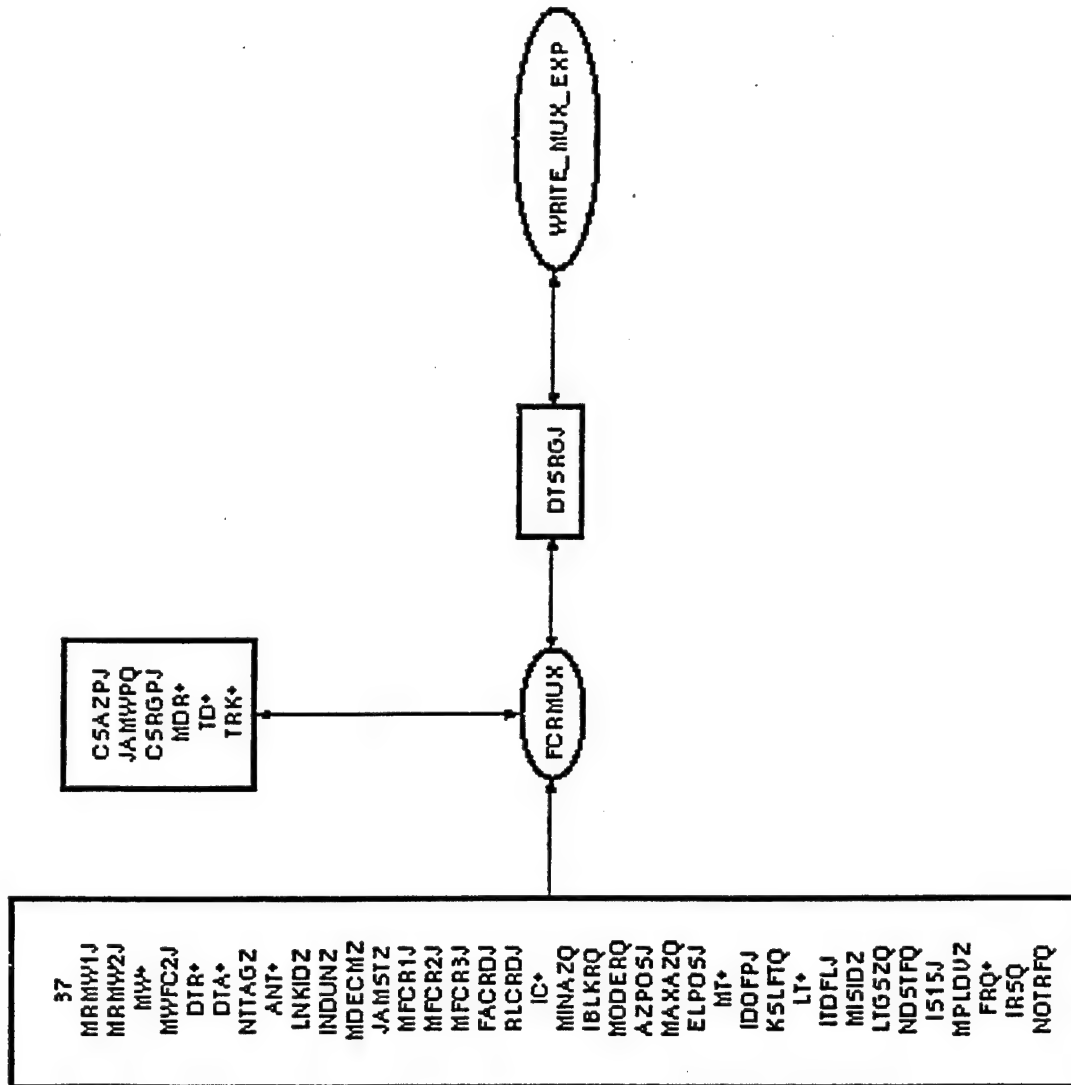


FCRSTT



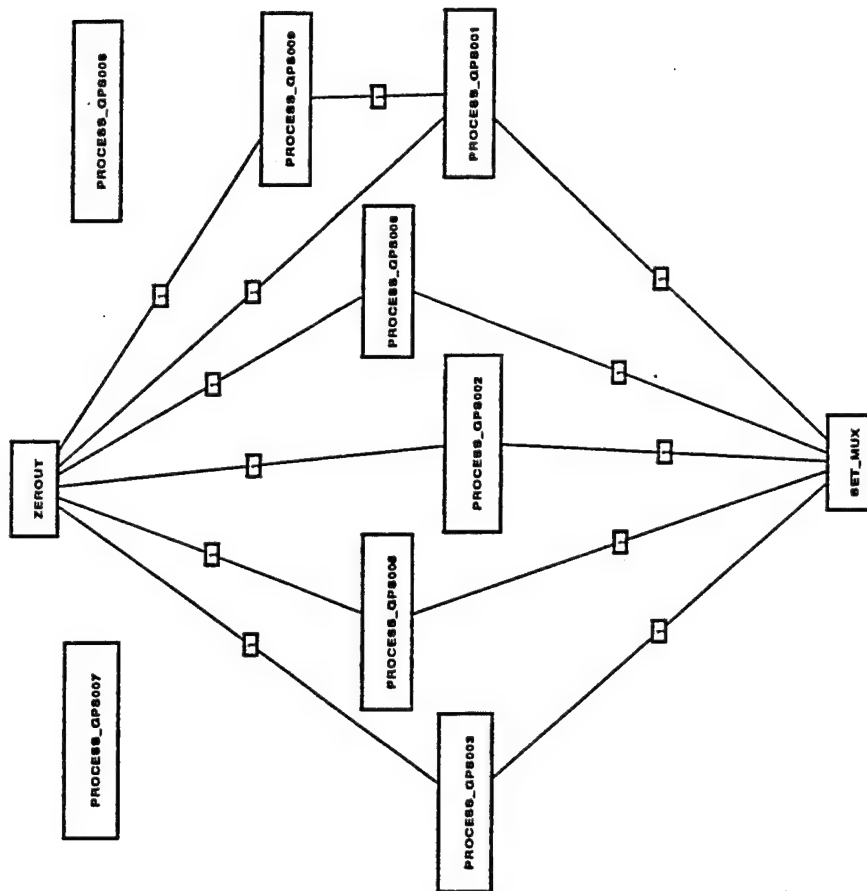


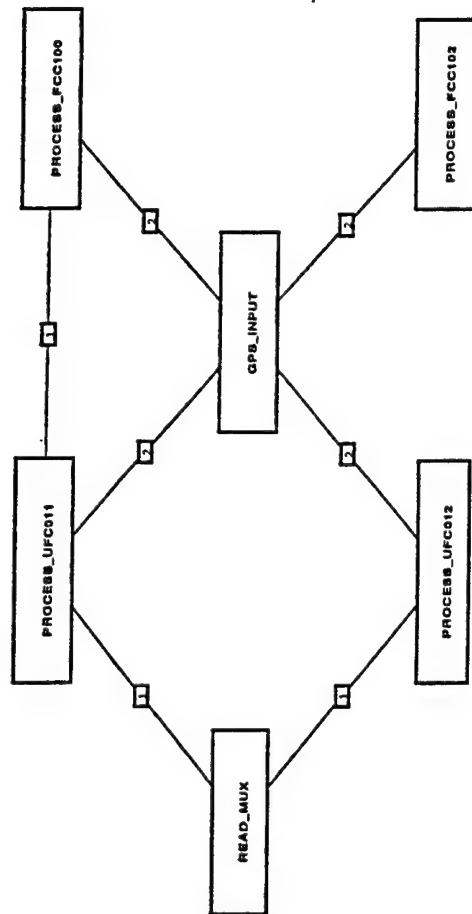
FCRMUX

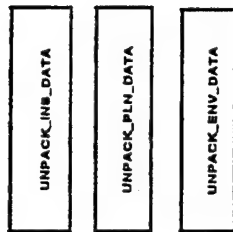


APPENDIX C

EXHIBIT GPS-P GPS Subsystem Packager Views



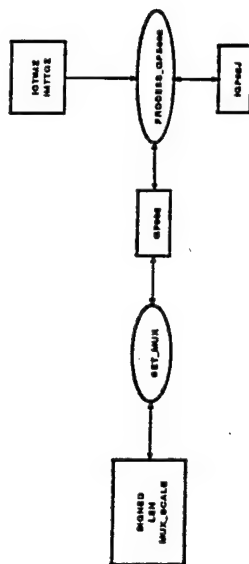


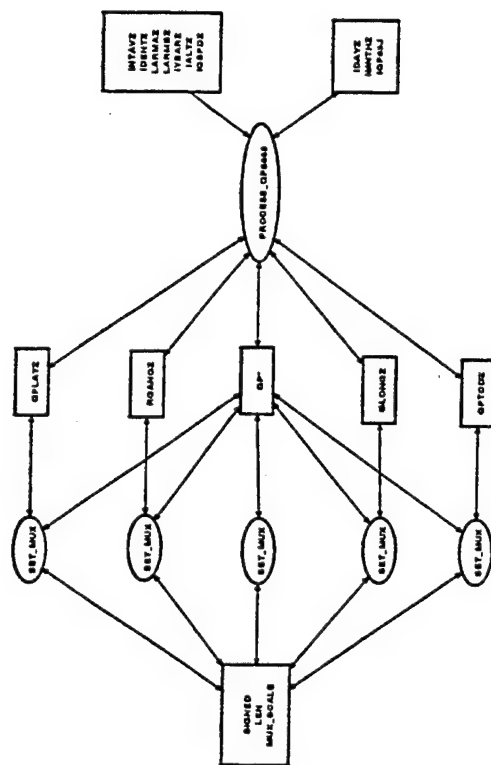


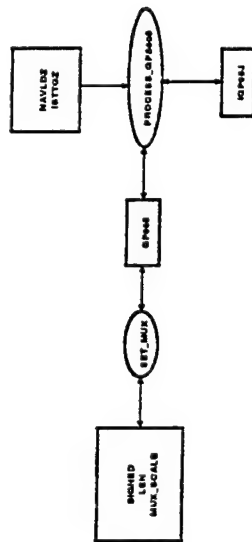
APPENDIX D

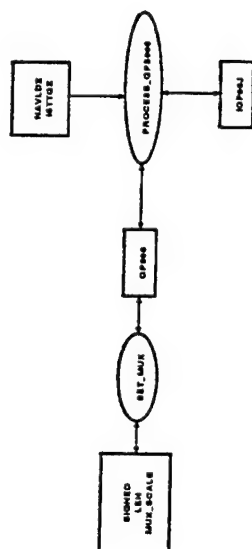
EXHIBIT GPS-D GPS Subsystem DFD Views

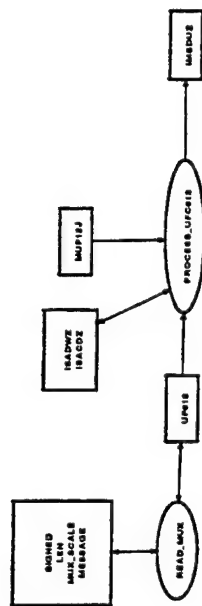










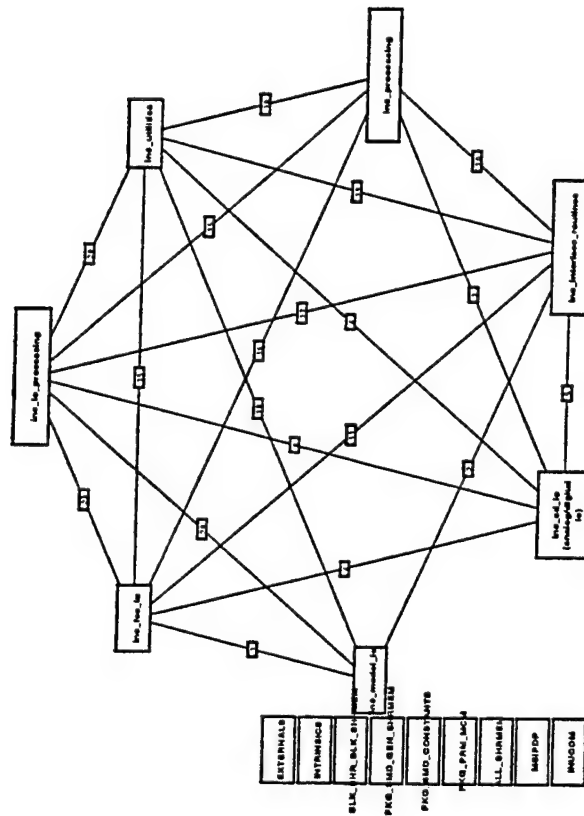


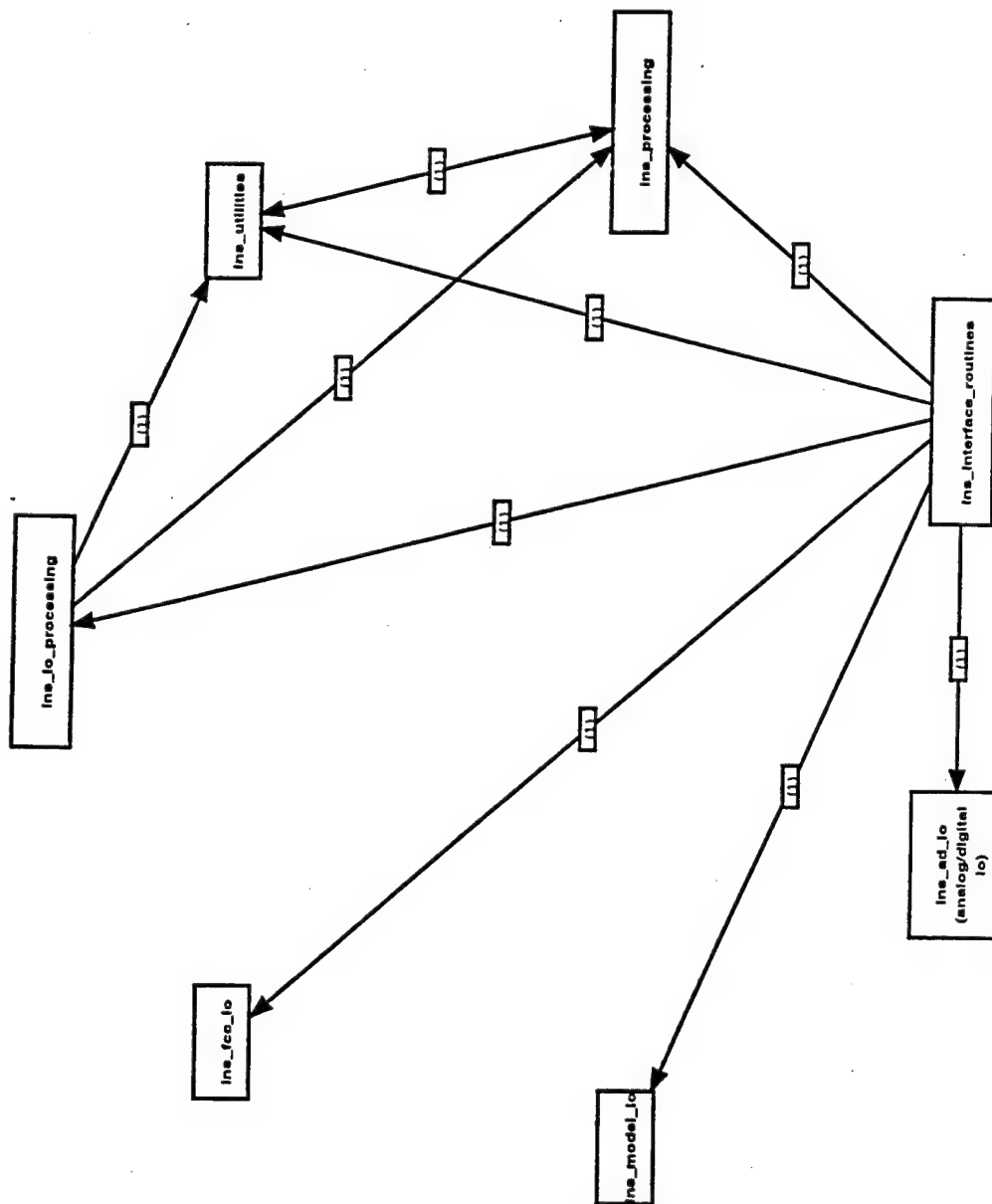


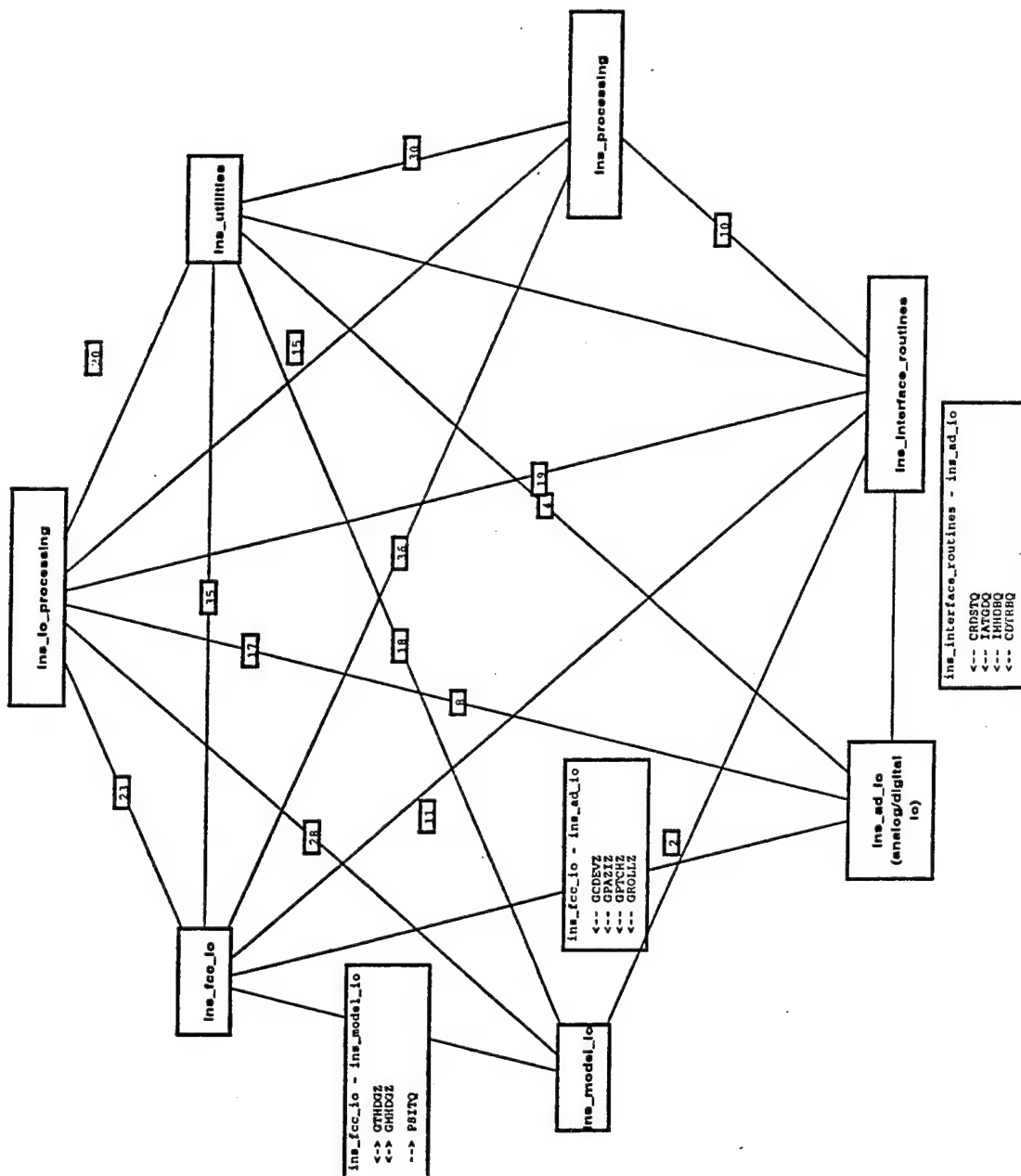
APPENDIX E

EXHIBIT INS-P

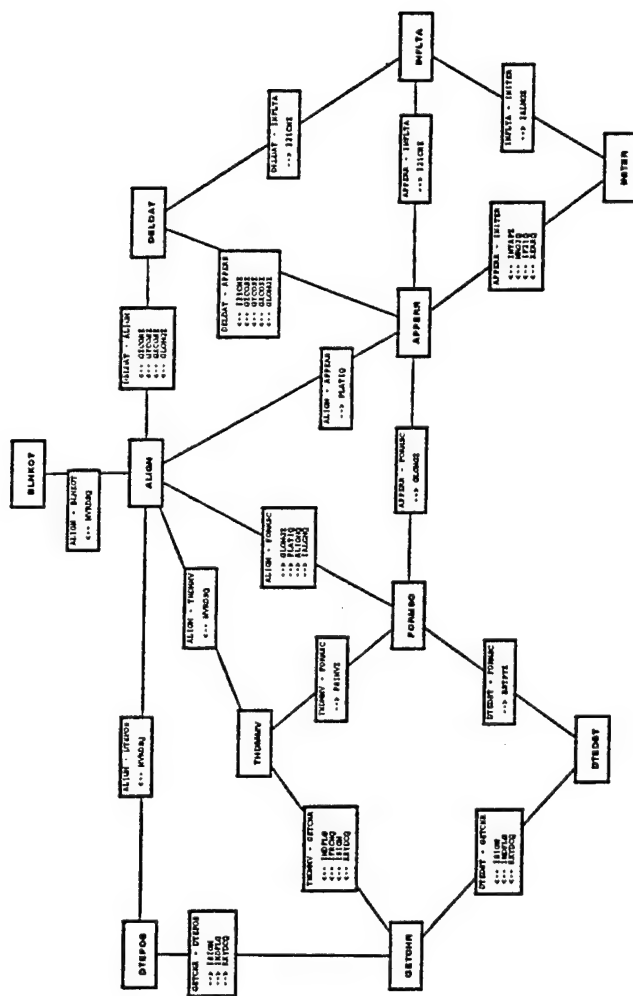
INS Subsystem Packager Views



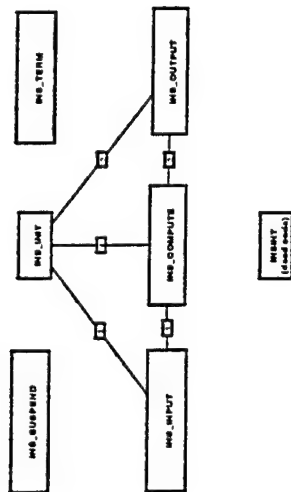




INSING	INSELO	INSING	INSEOR
INSER	INSER	INSEOR	INSEAT

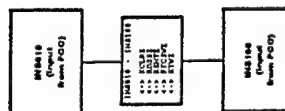


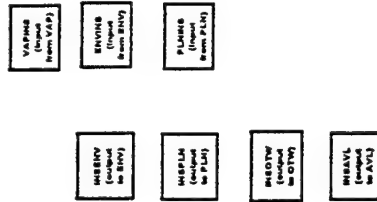




INSTR
(analog/digital
input)

INSTR
(analog/digital
input)



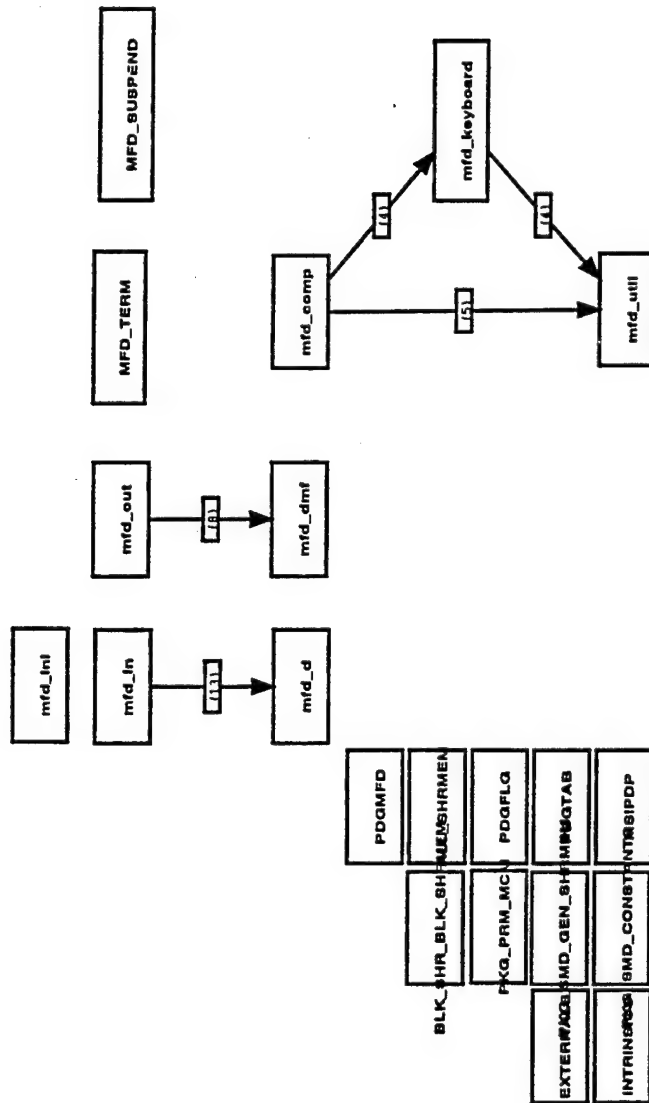


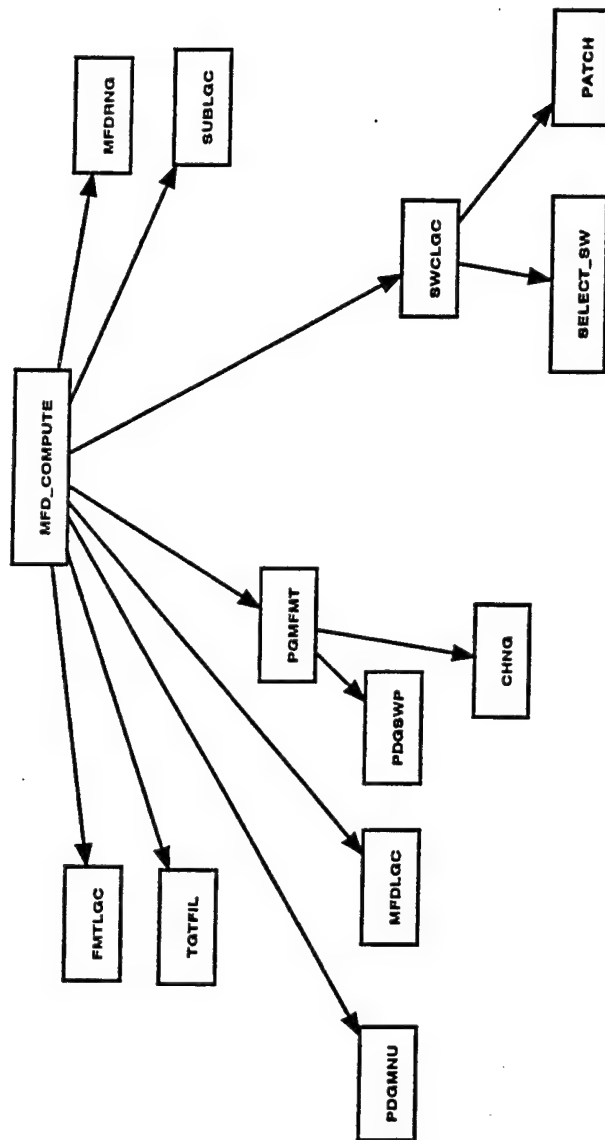
APPENDIX F

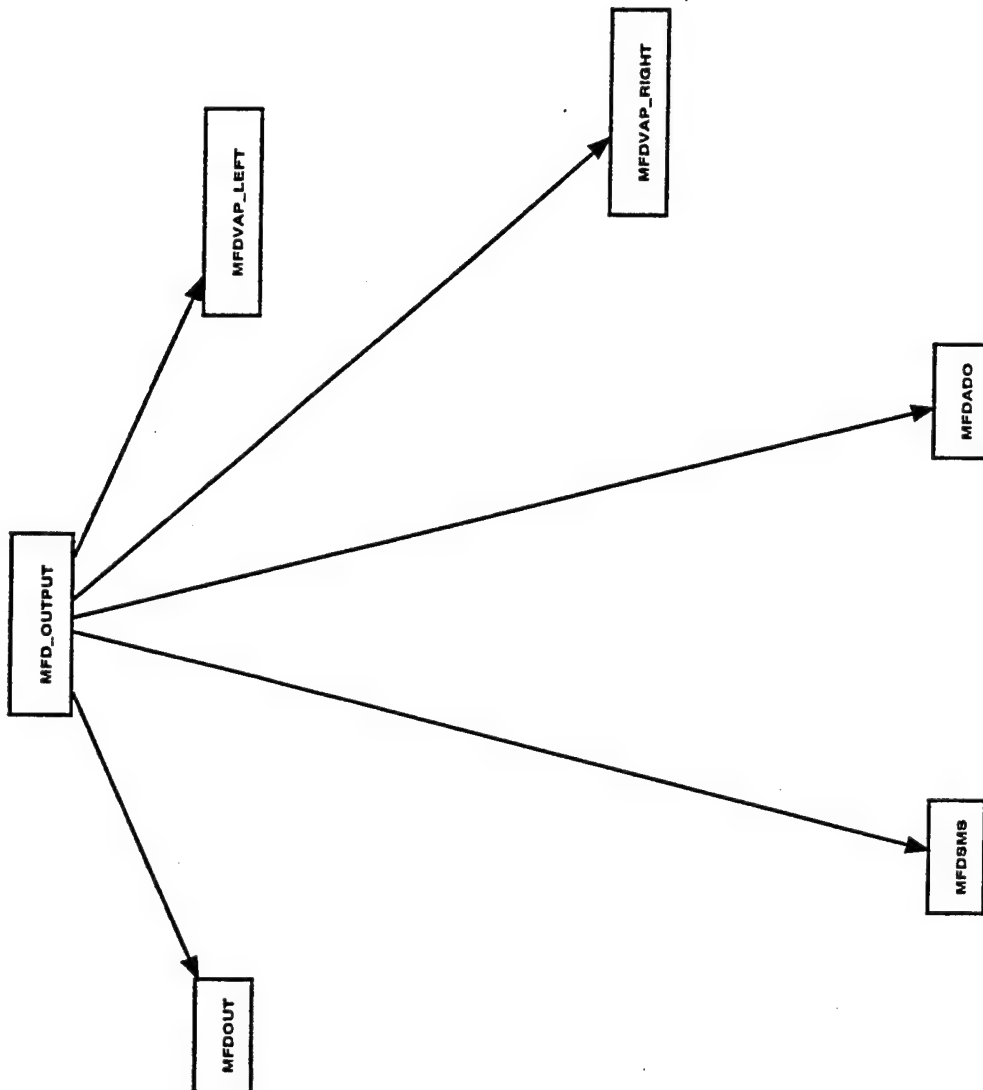
EXHIBIT INS-D INS Subsystem DFD Views

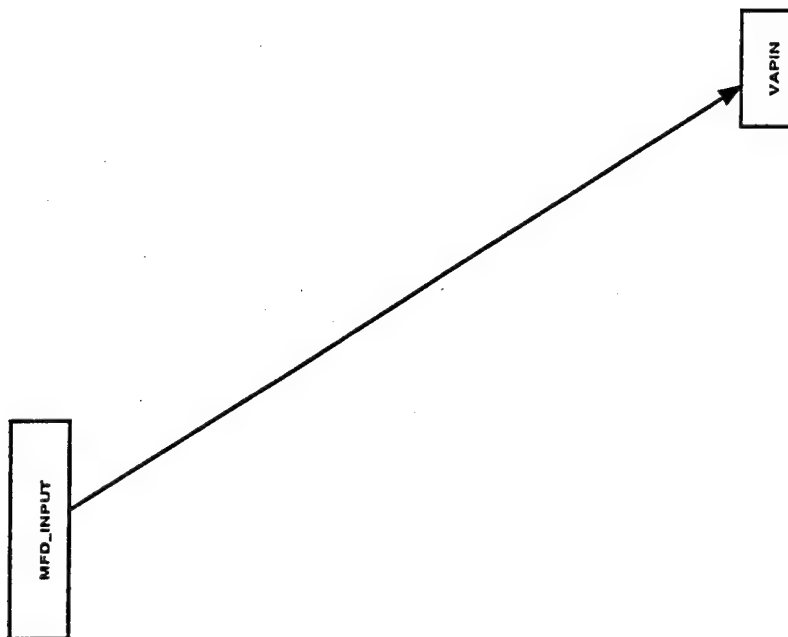
APPENDIX G

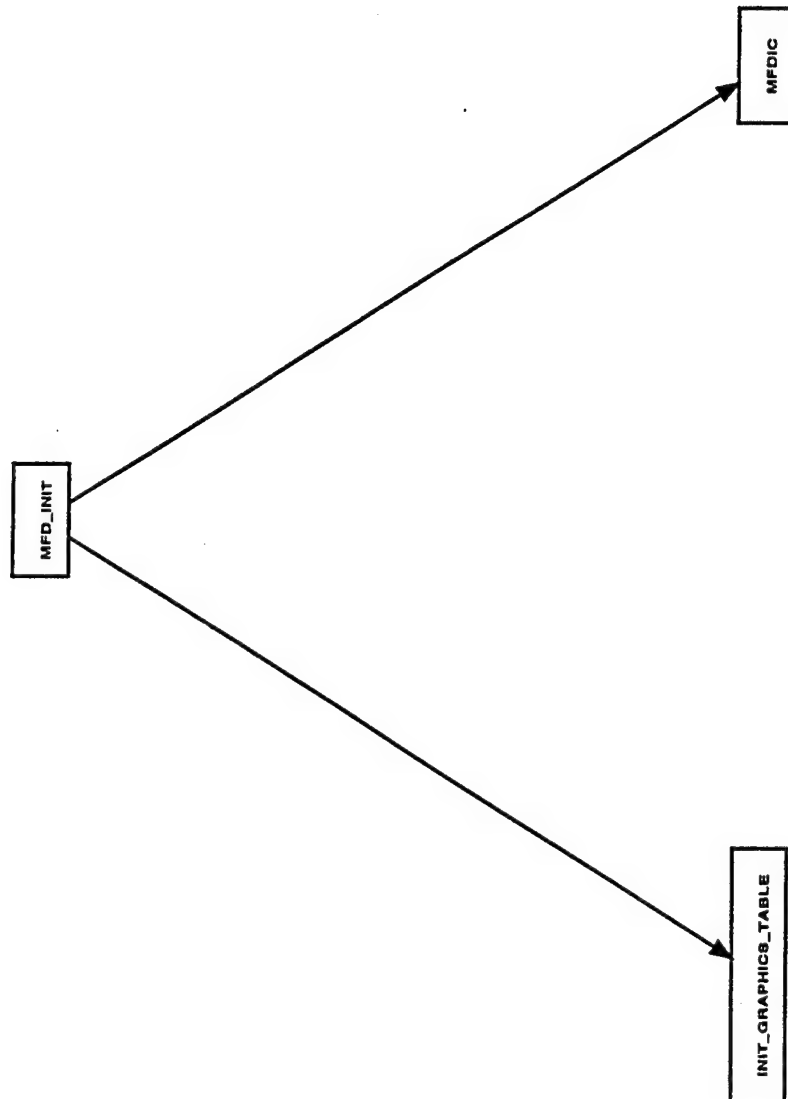
EXHIBIT MFD-P MFD Subsystem Packager Views









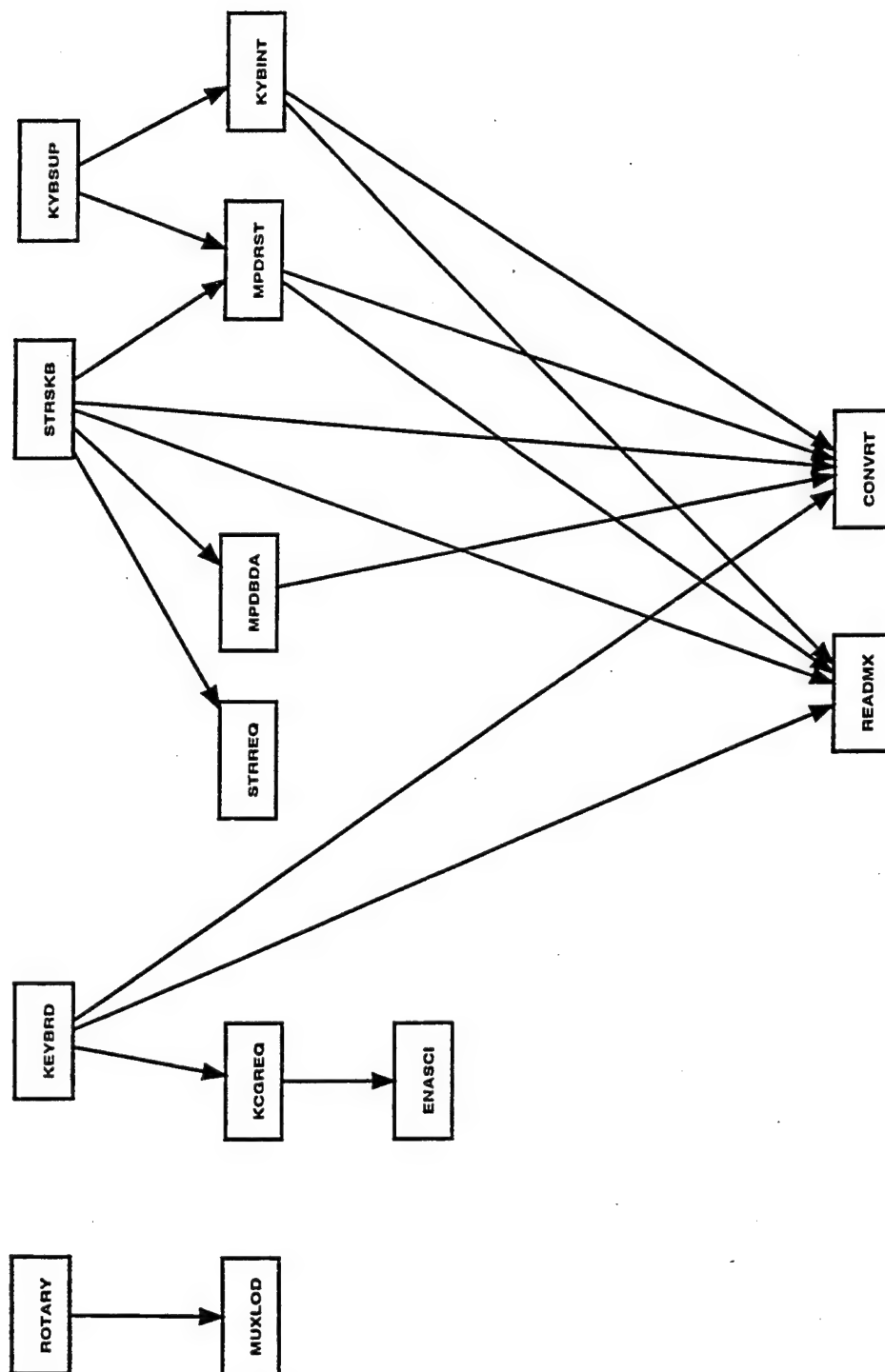


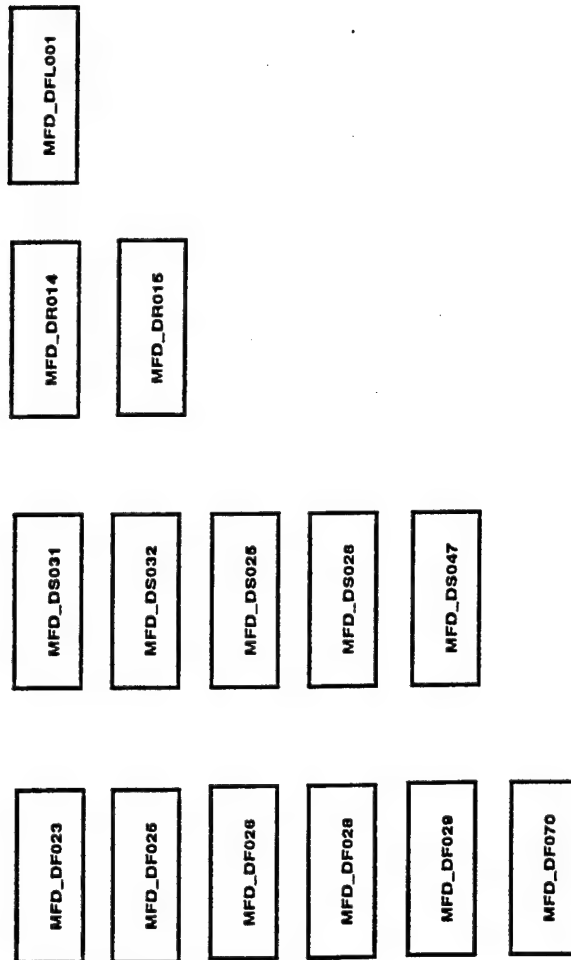
MPDNUM

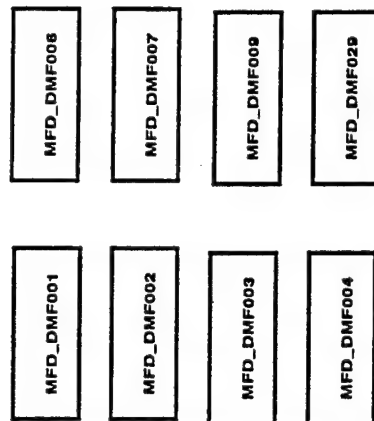
CHGREQ

ROTATE



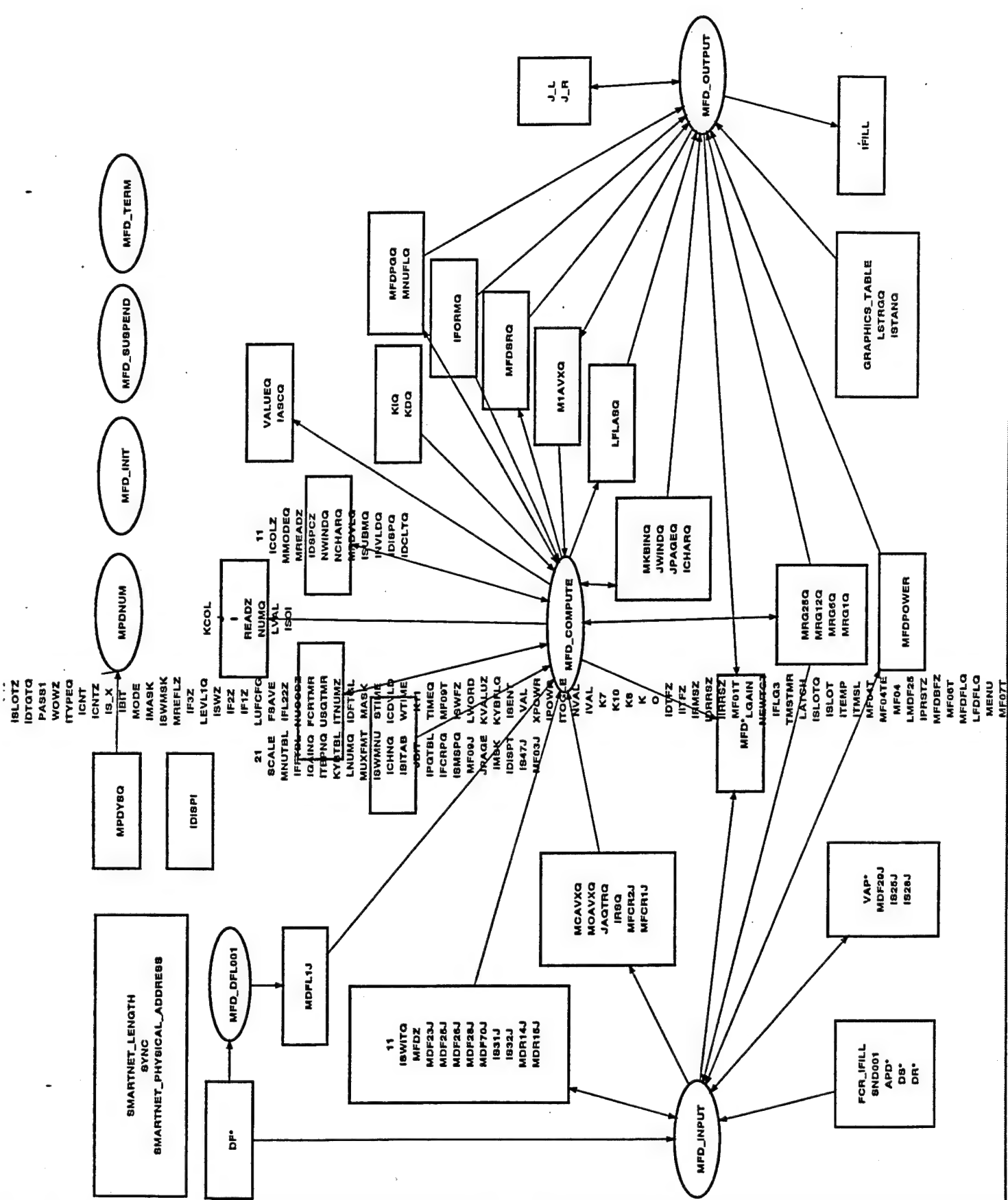


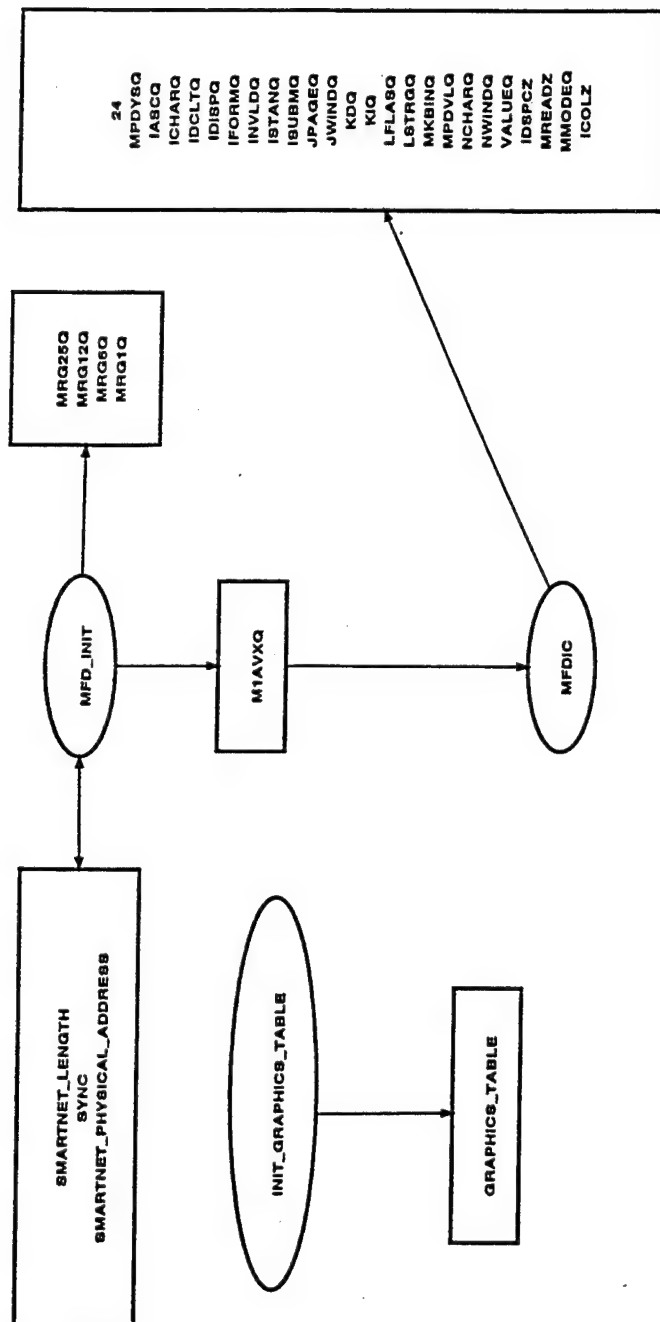


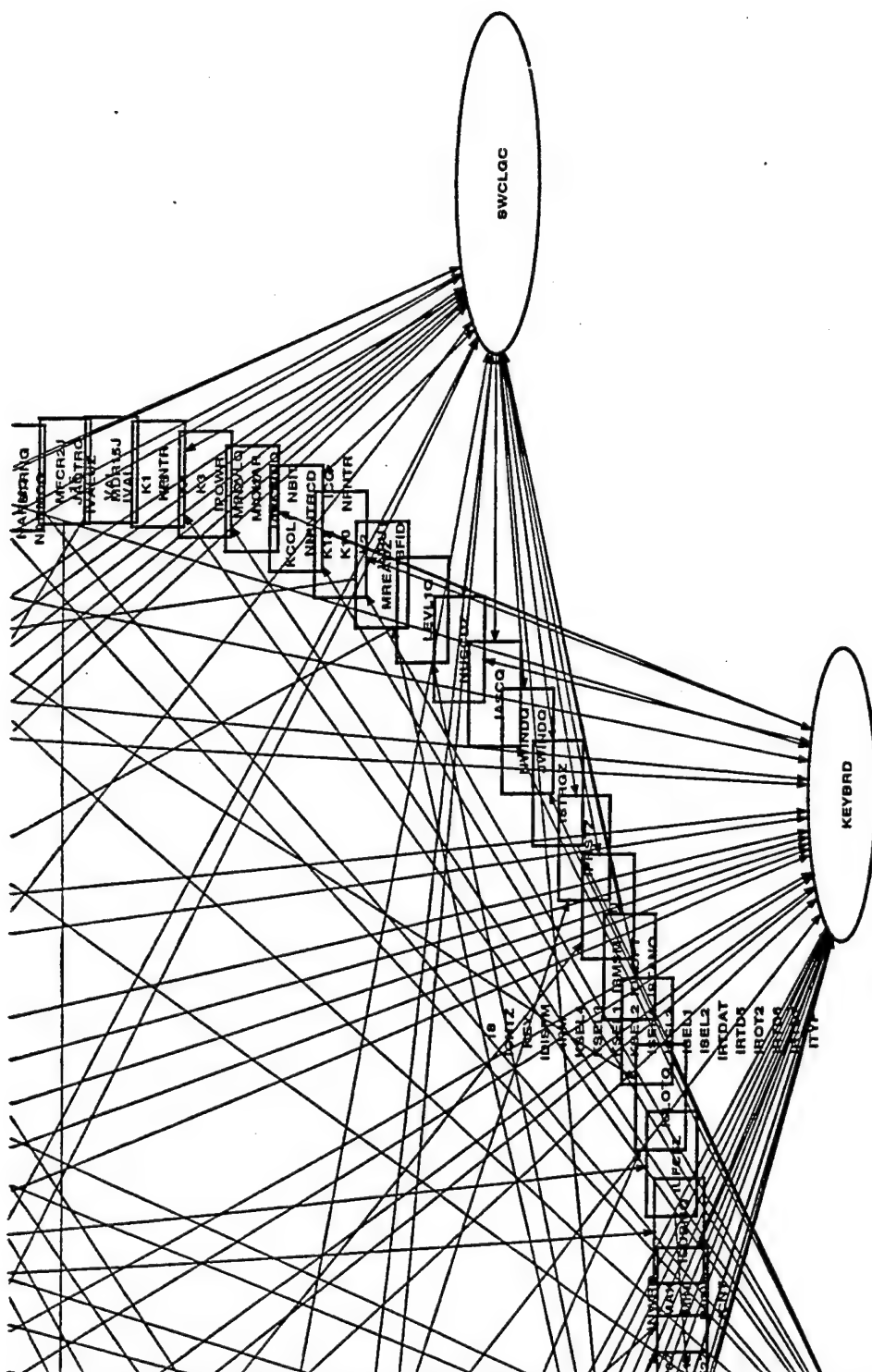


APPENDIX H

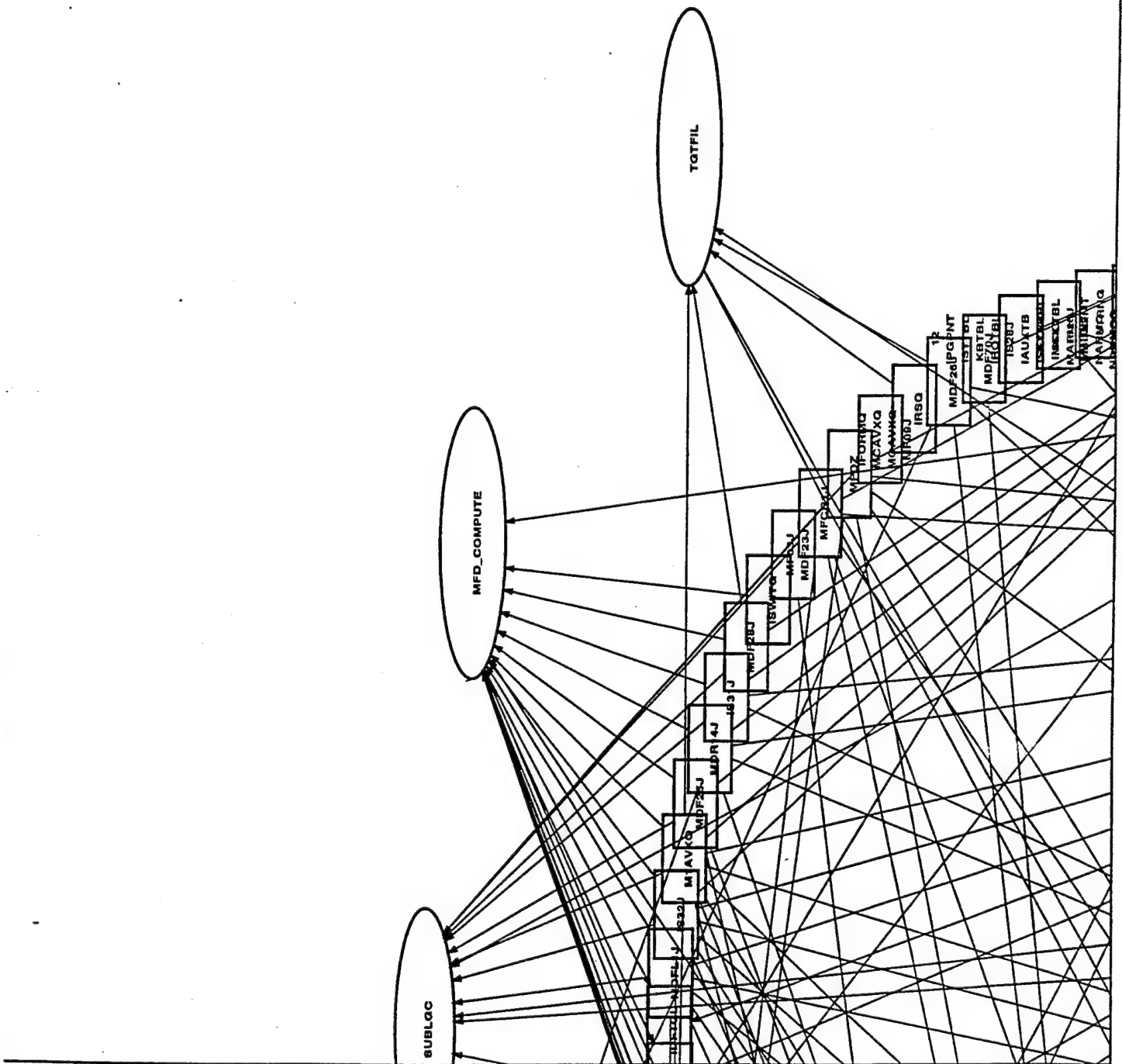
EXHIBIT MFD-D MFD Subsystem DFD Views



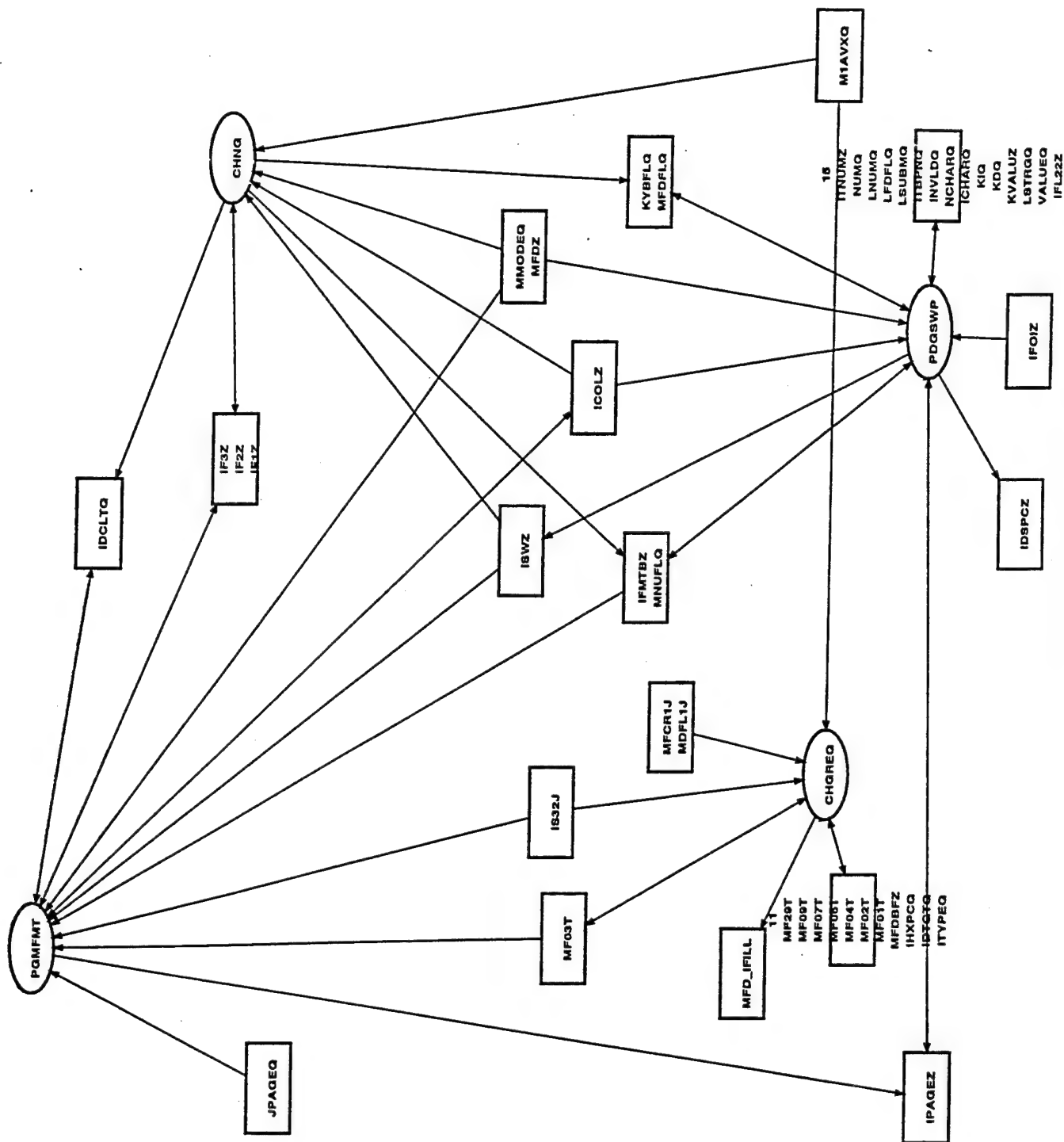


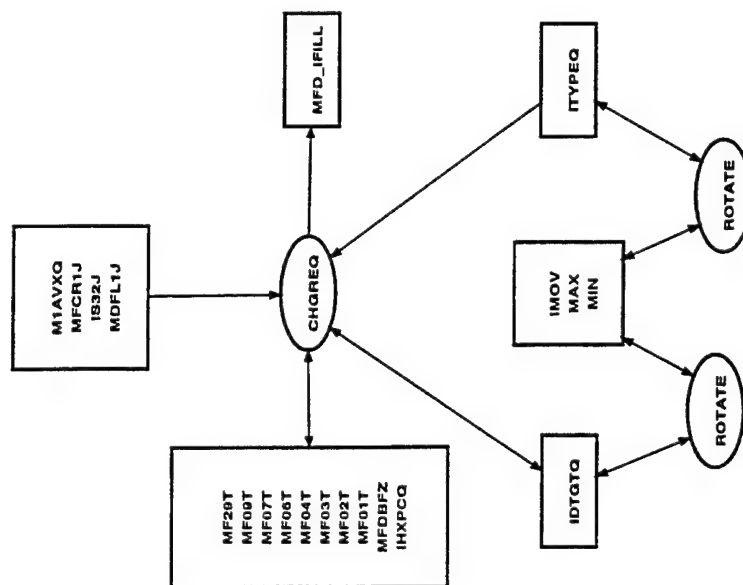


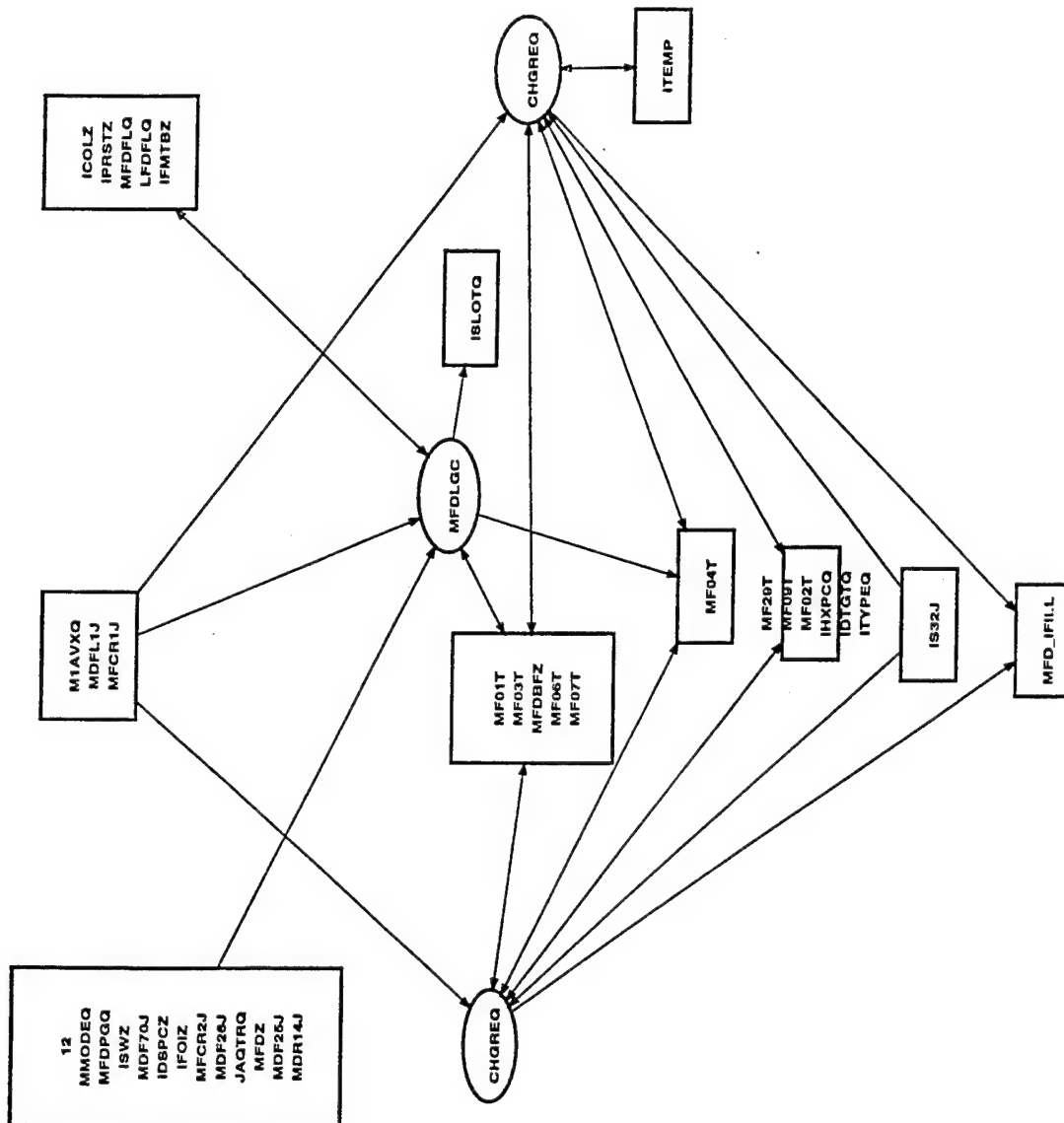
Data Flow Diagram - MFD_COMPUTE 0 - 1



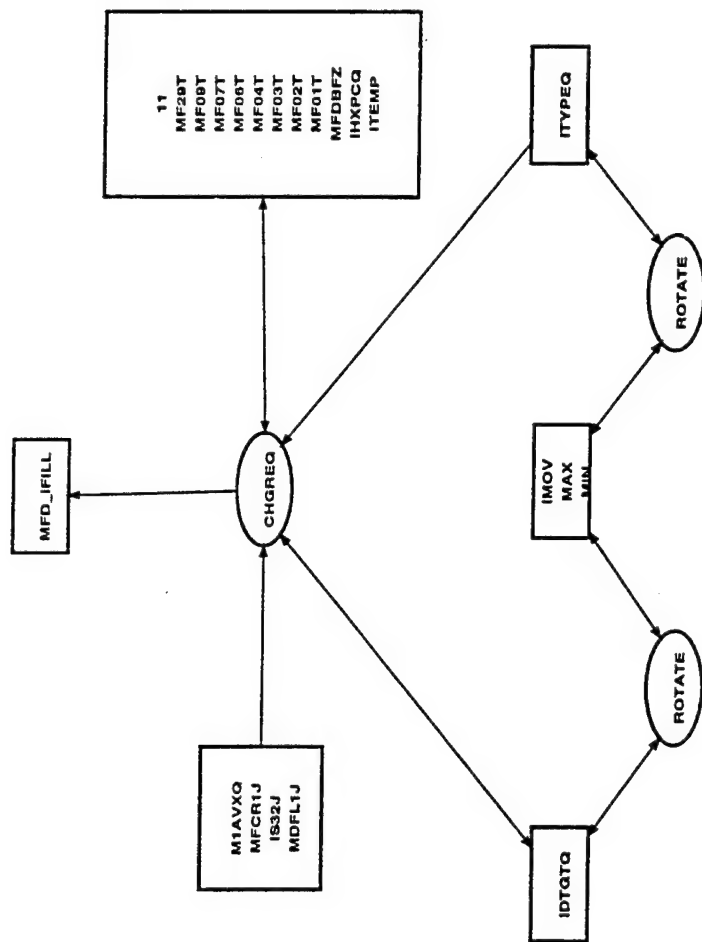
Data Flow Diagram - MFD_COMPUTE 1 - 1

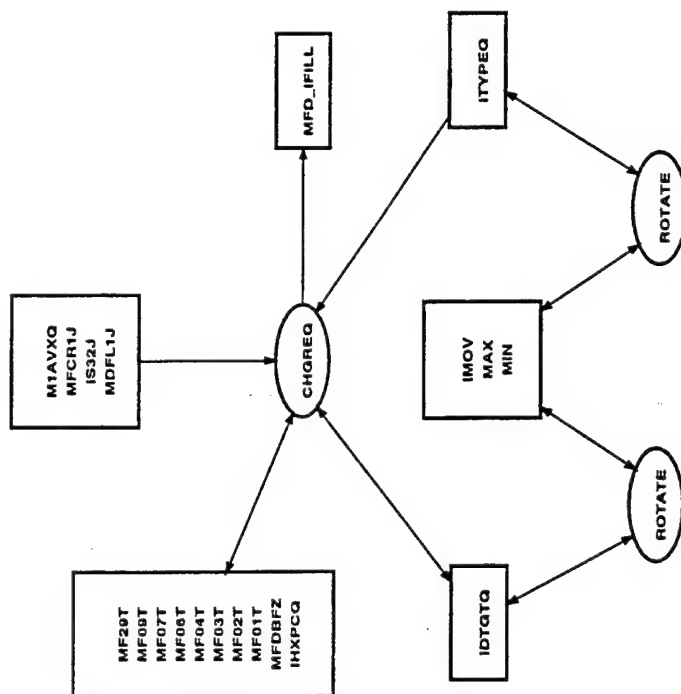


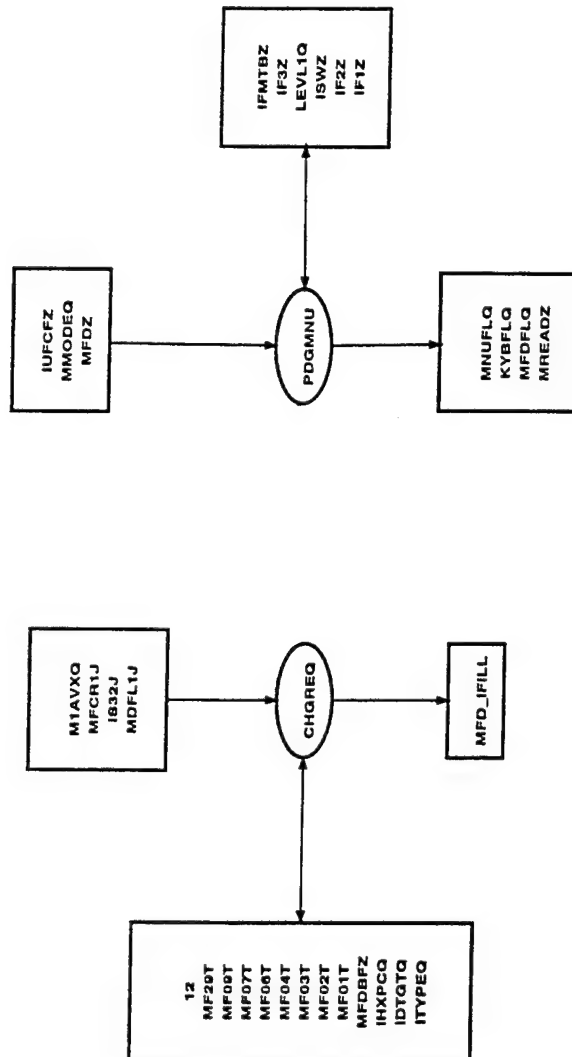


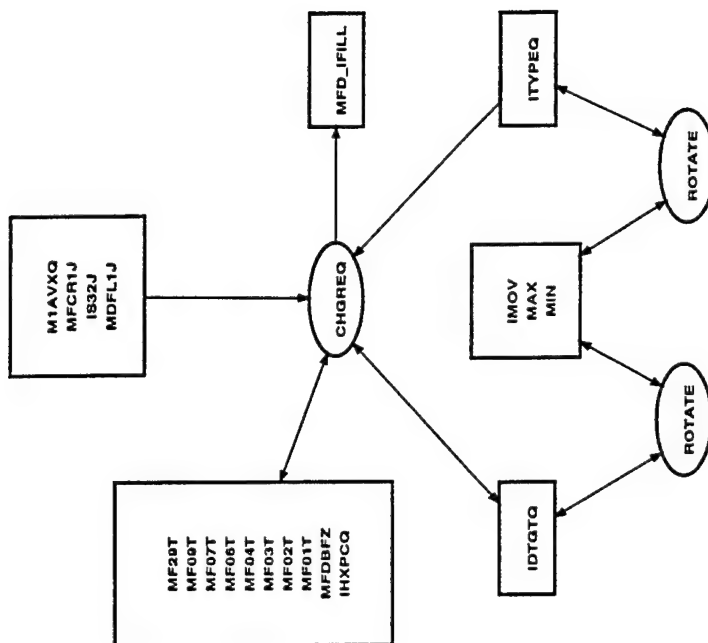


Data Flow Diagram - MFDLGC

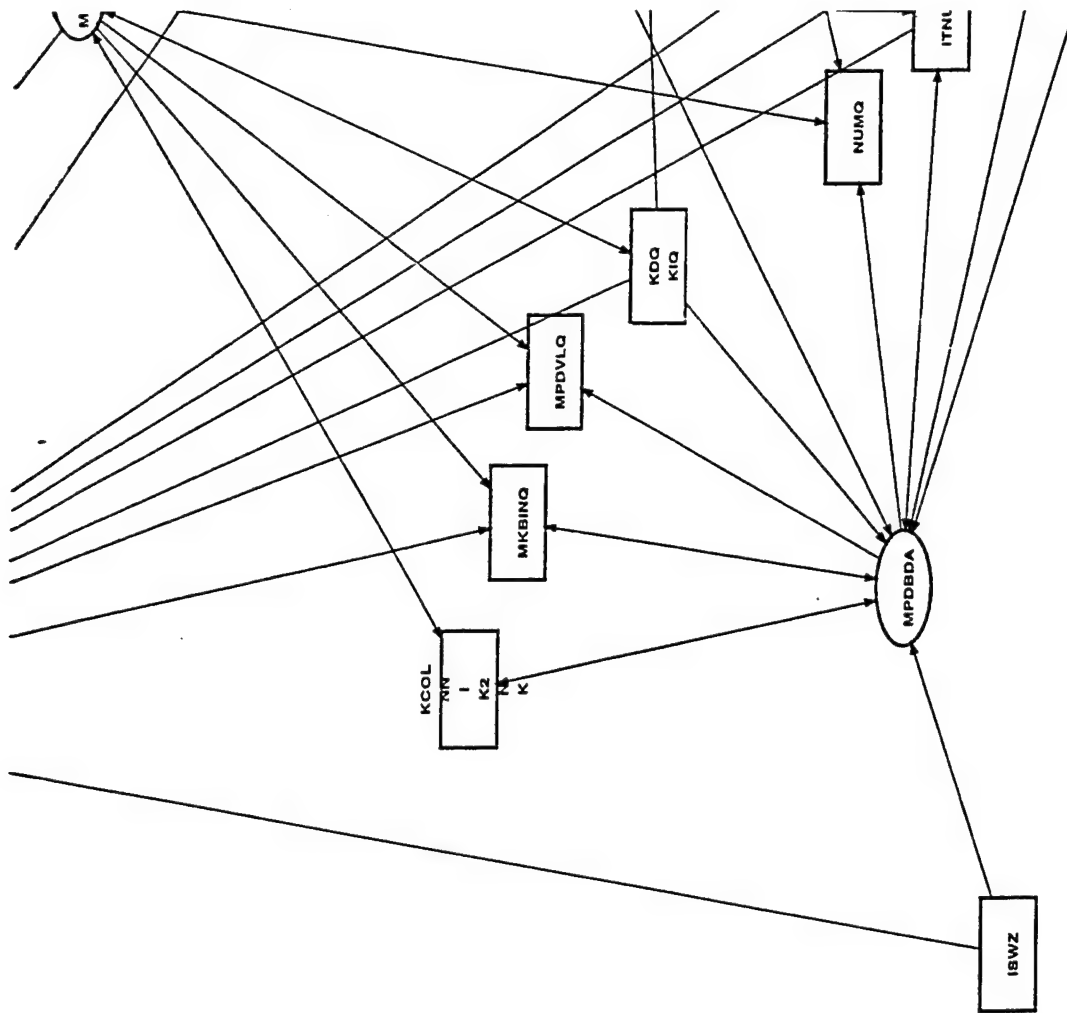


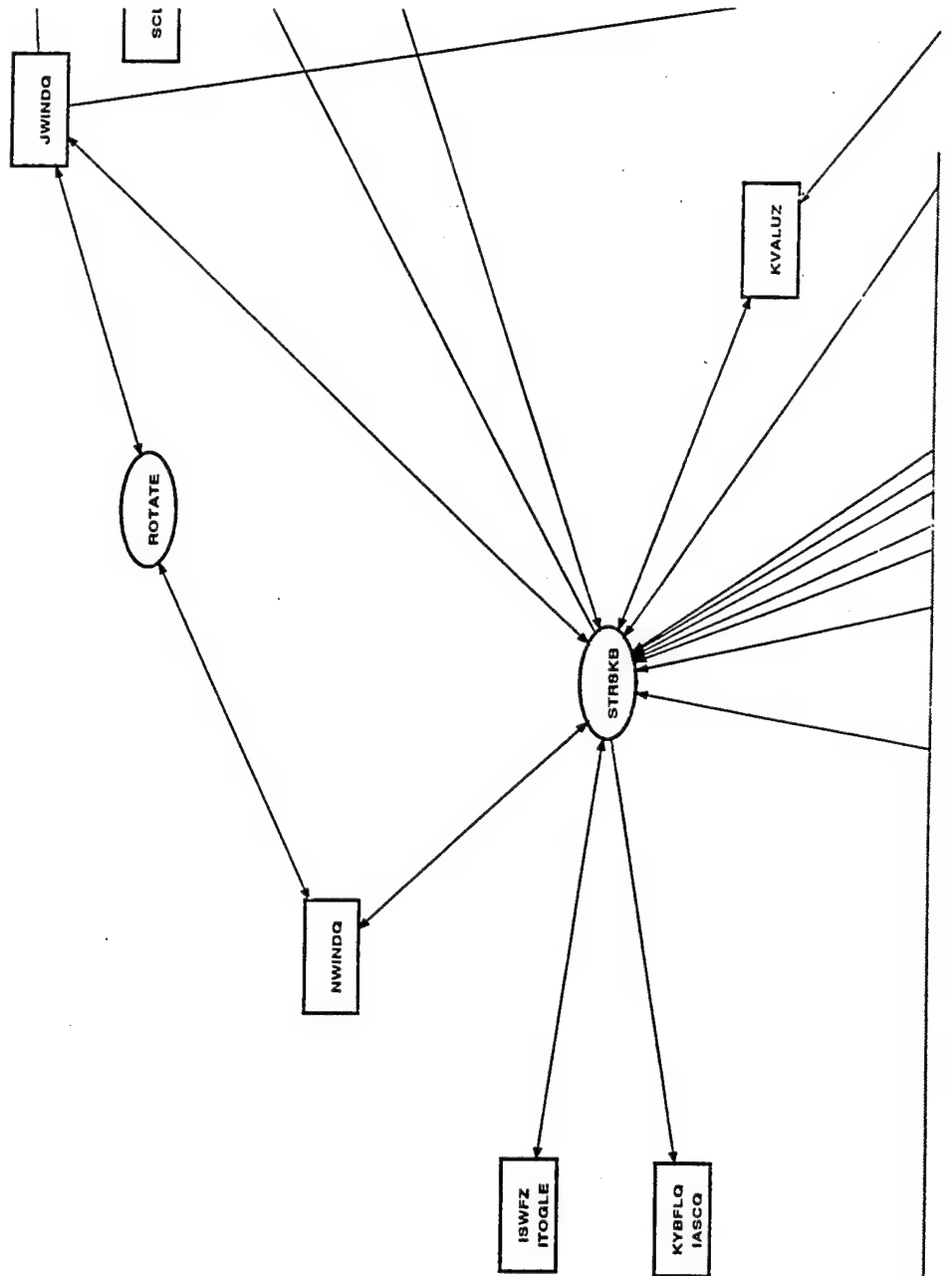


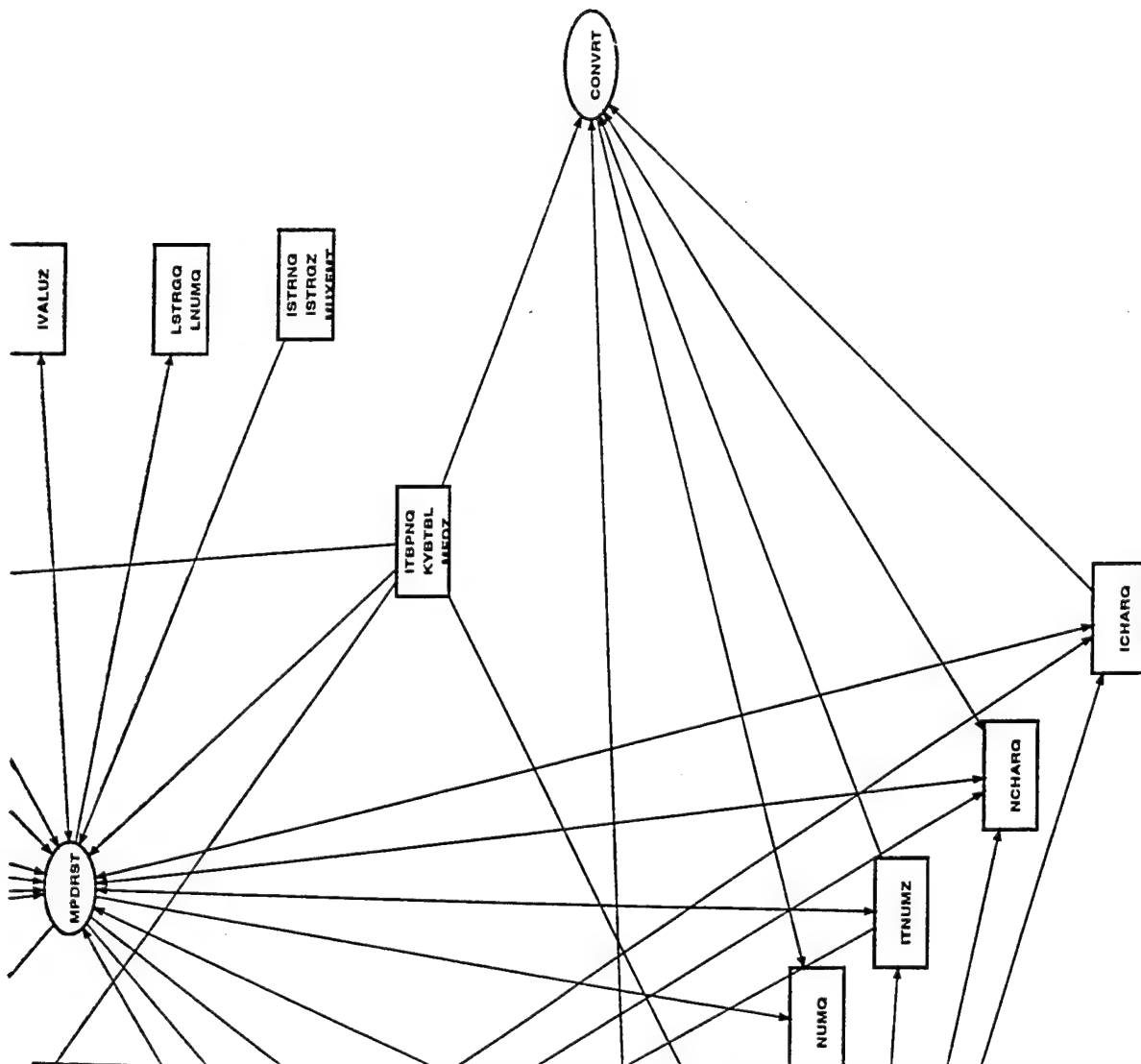




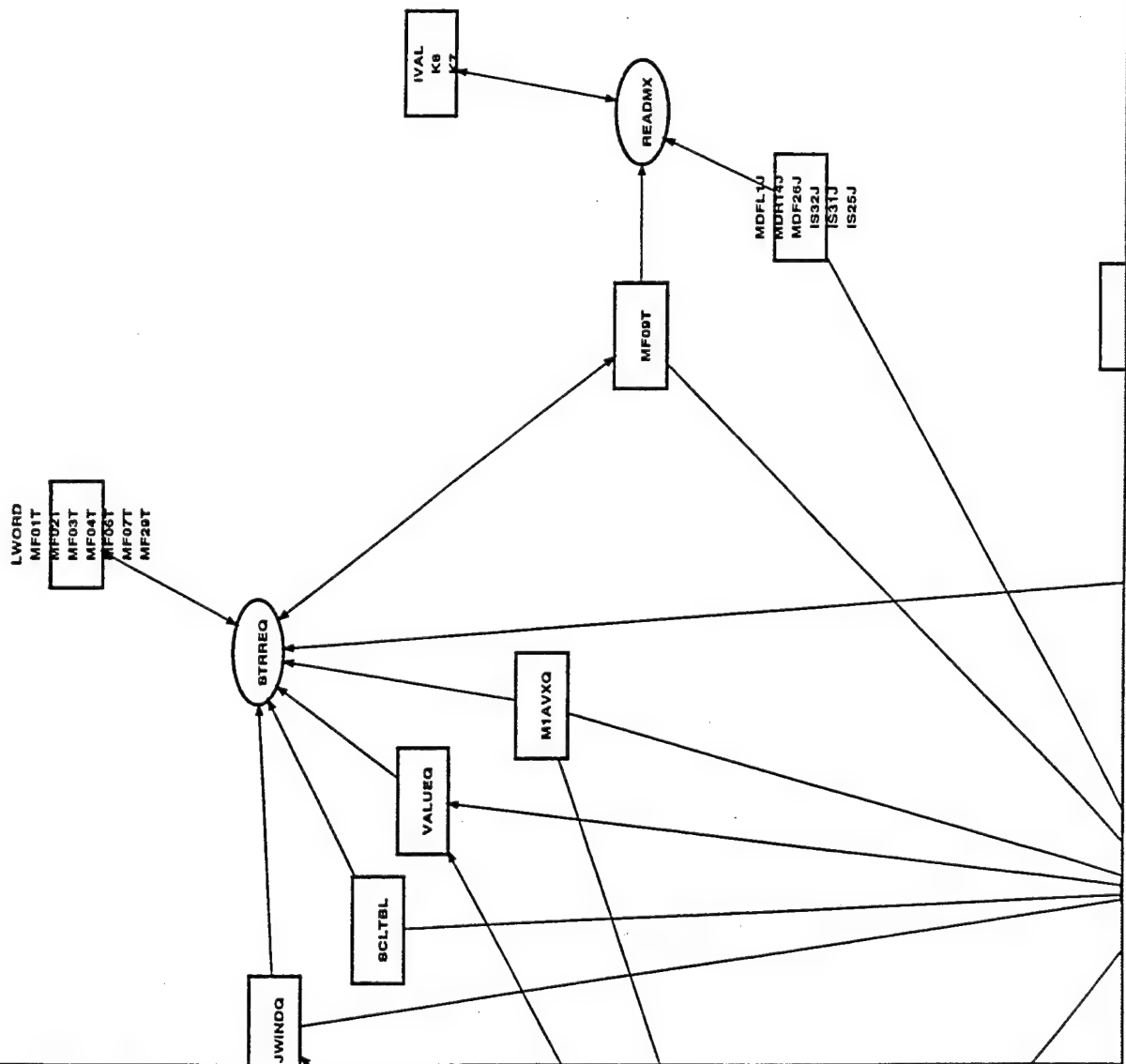
Data Flow Diagram - CHGREQ

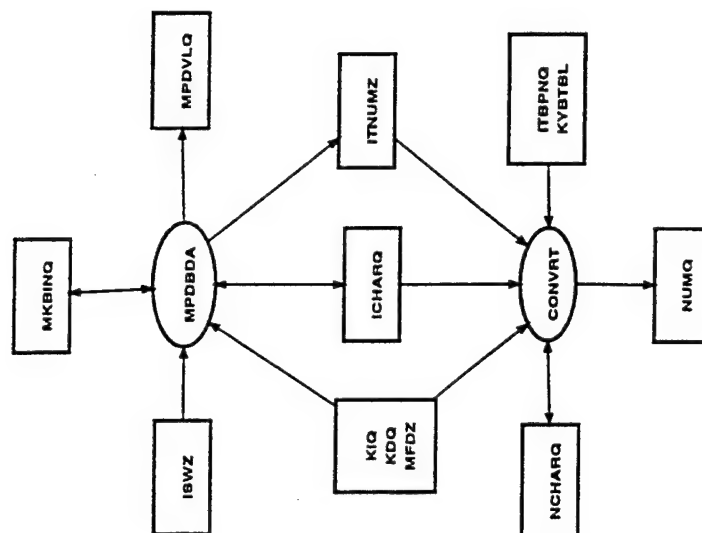


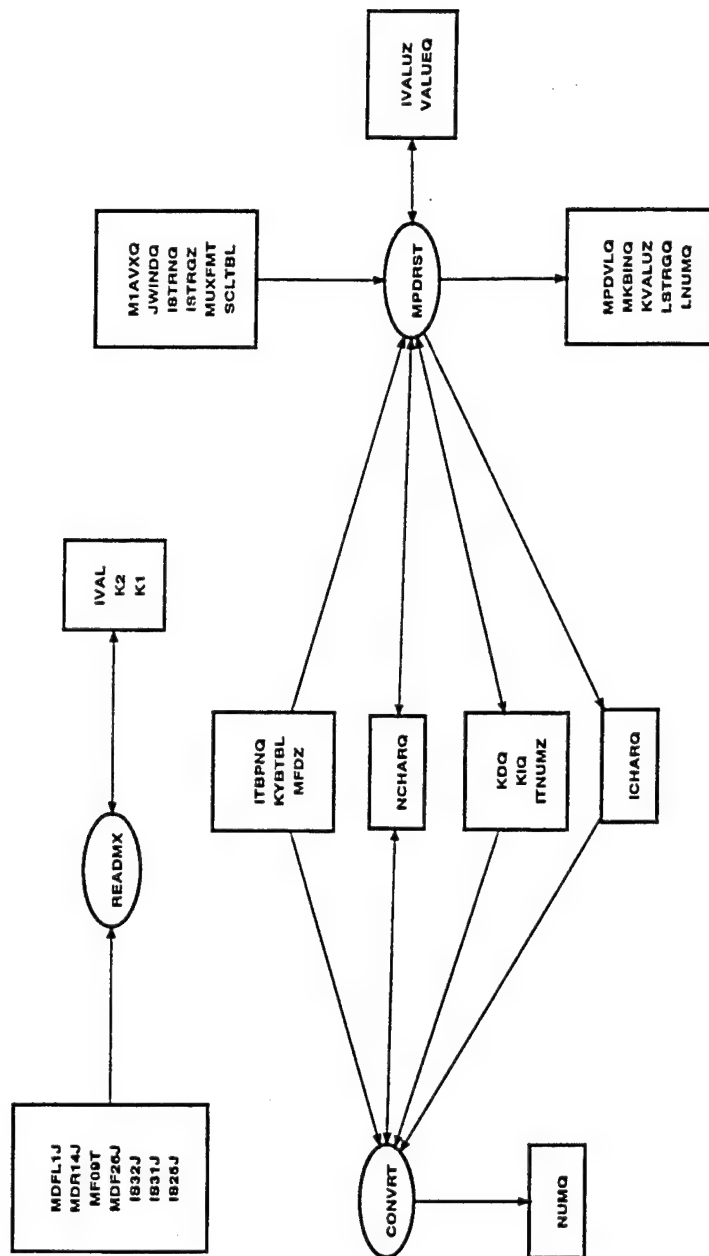


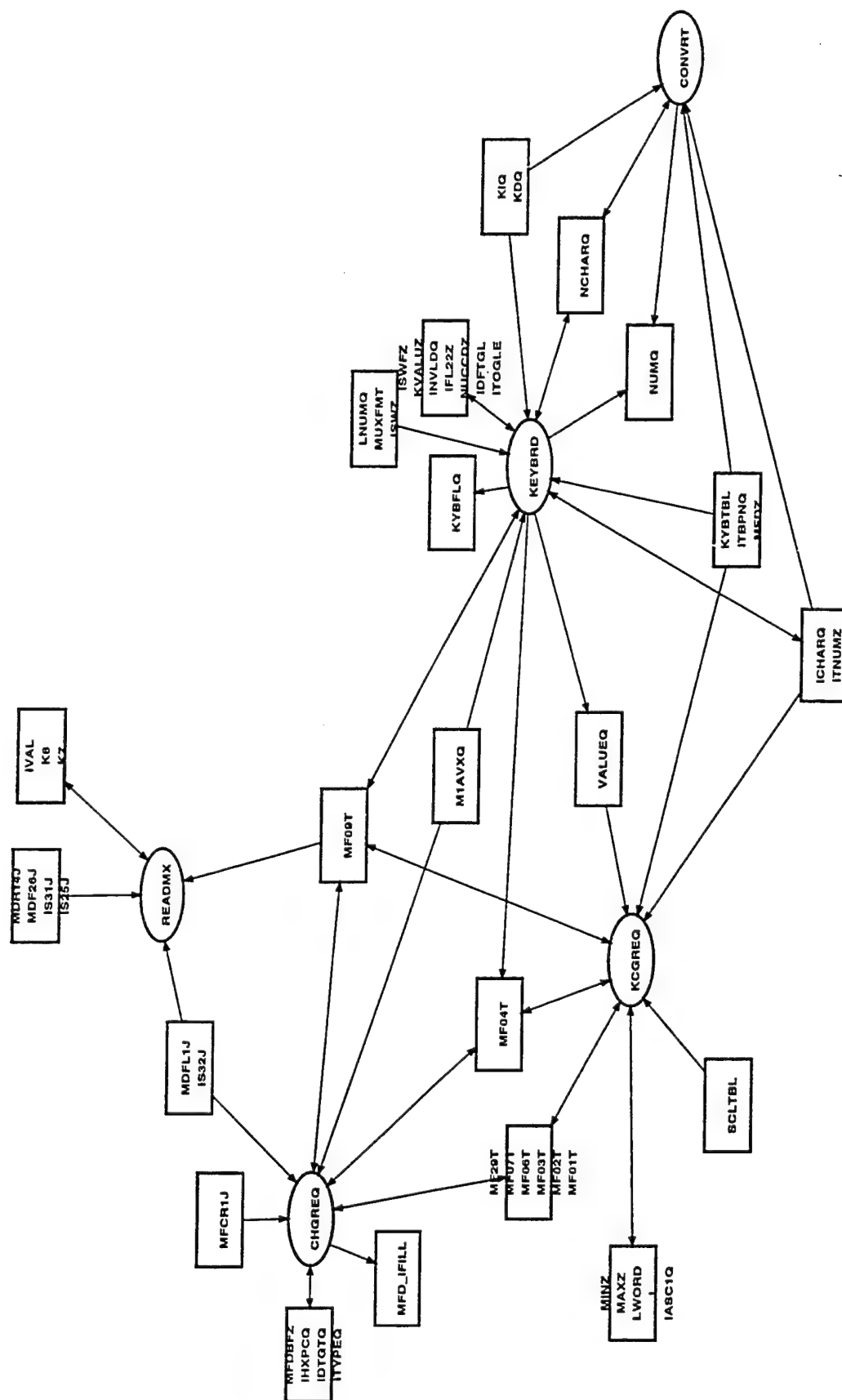


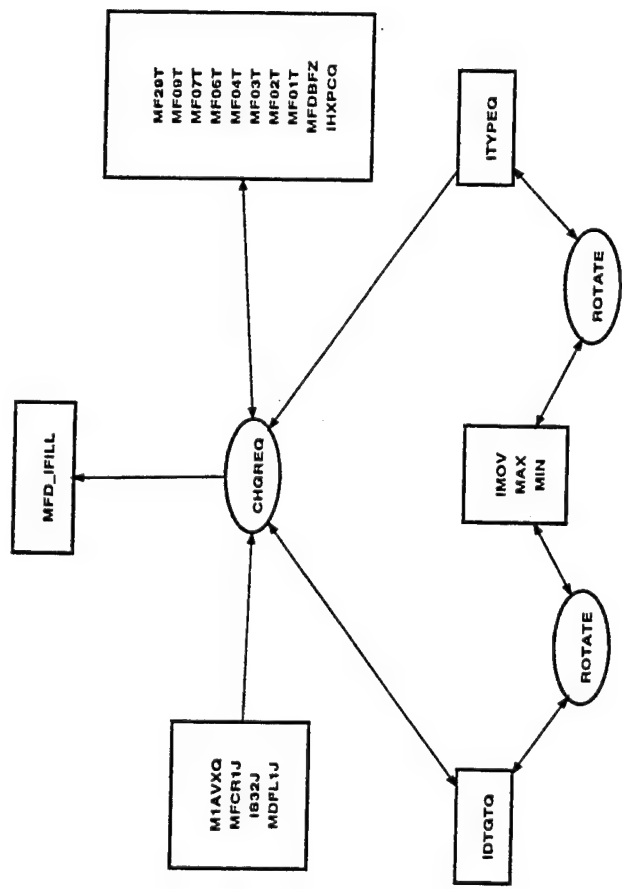
Data Flow Diagram - STRSKB 0--1

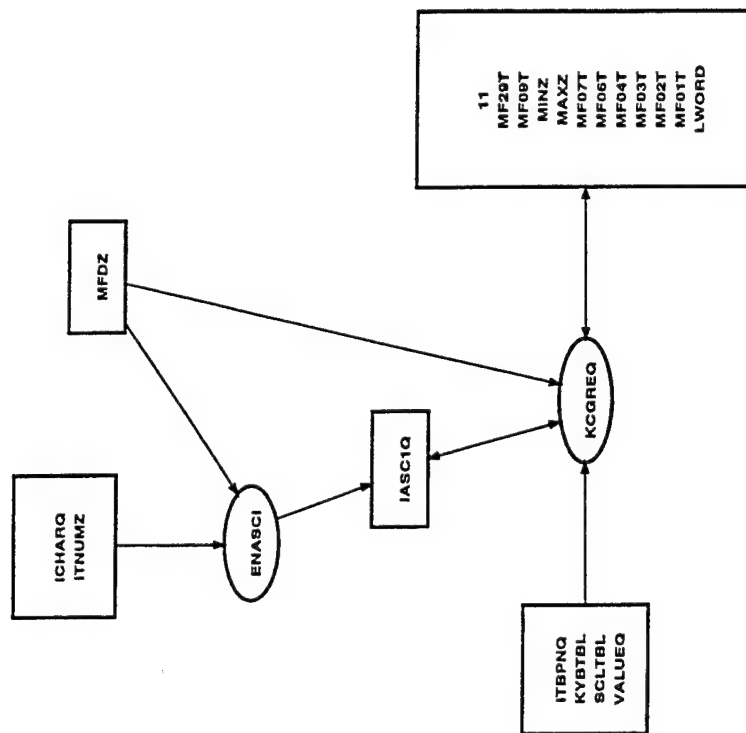


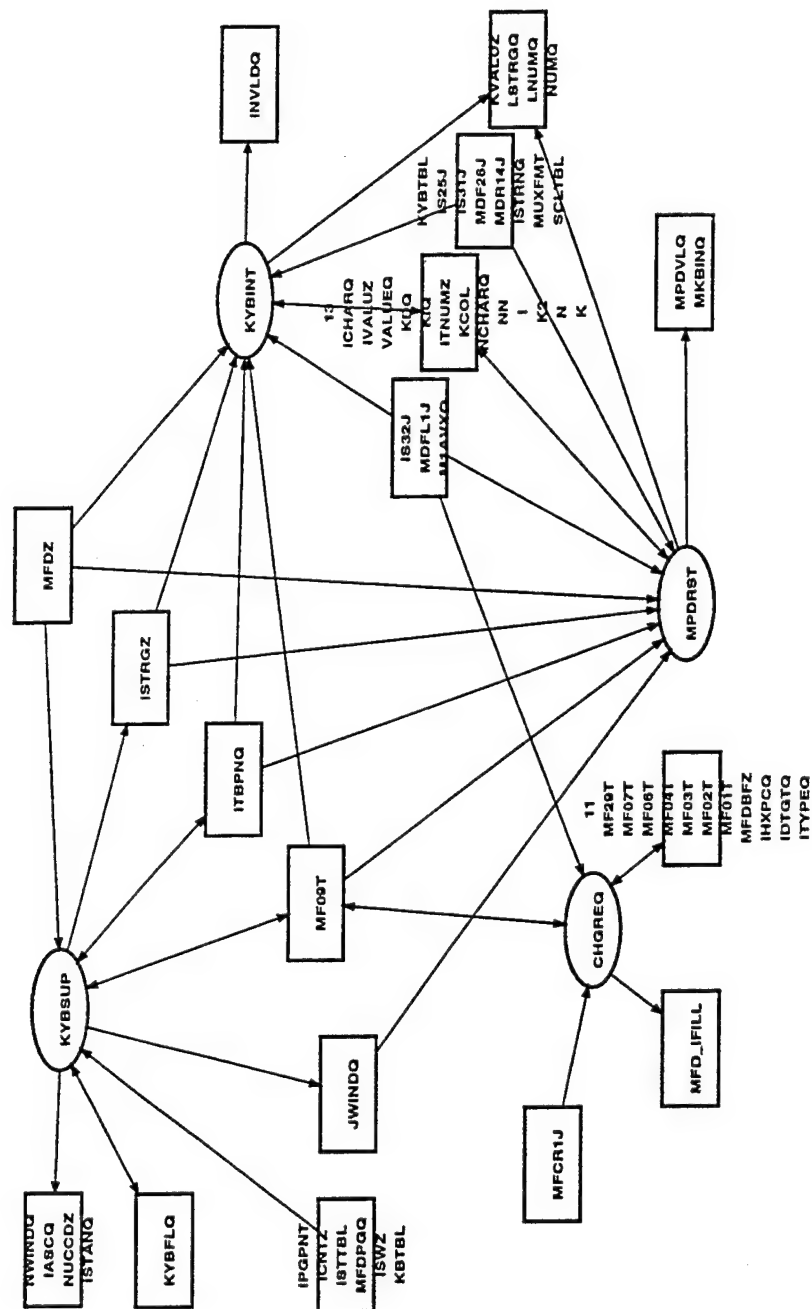


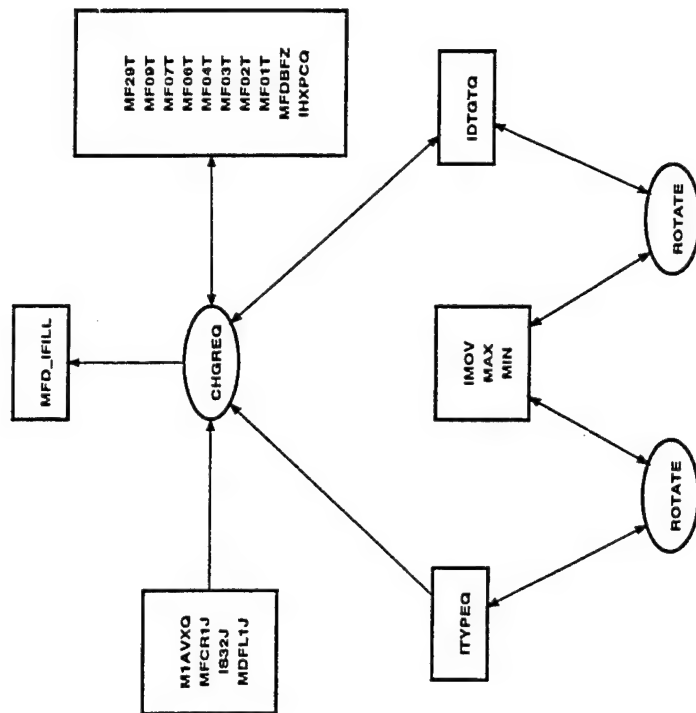


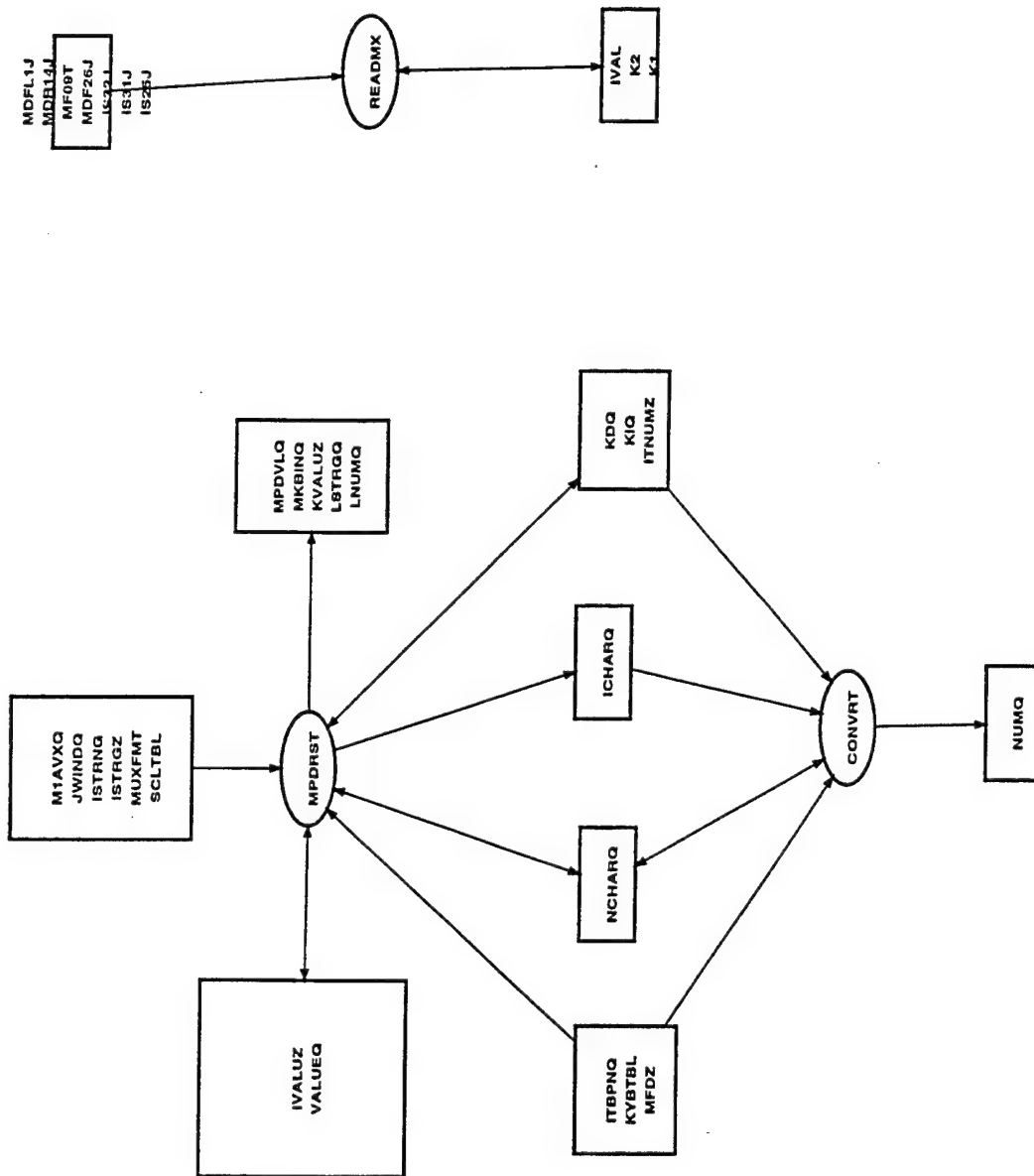


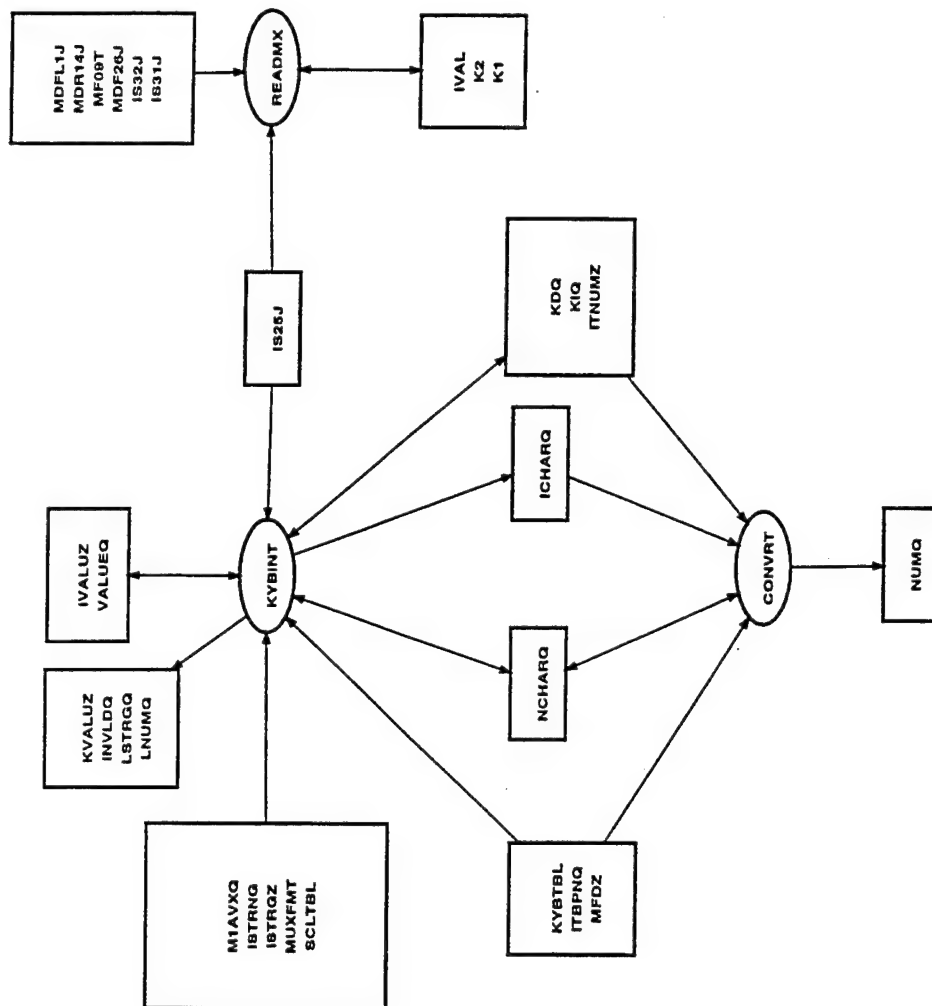


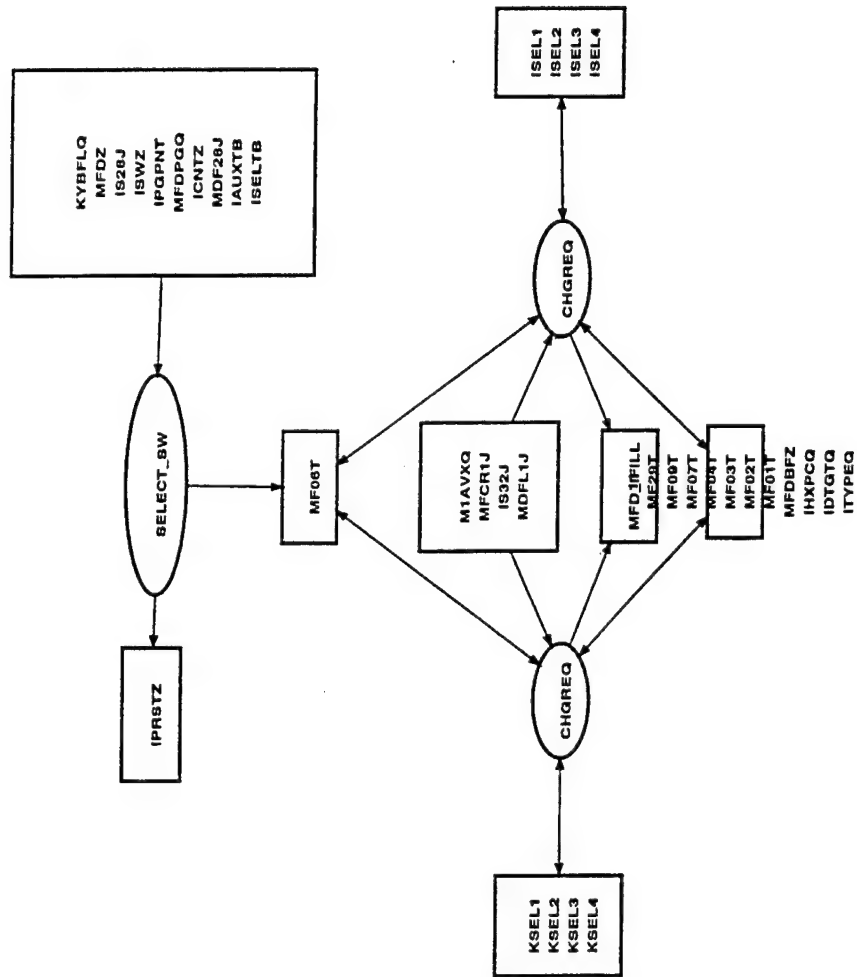


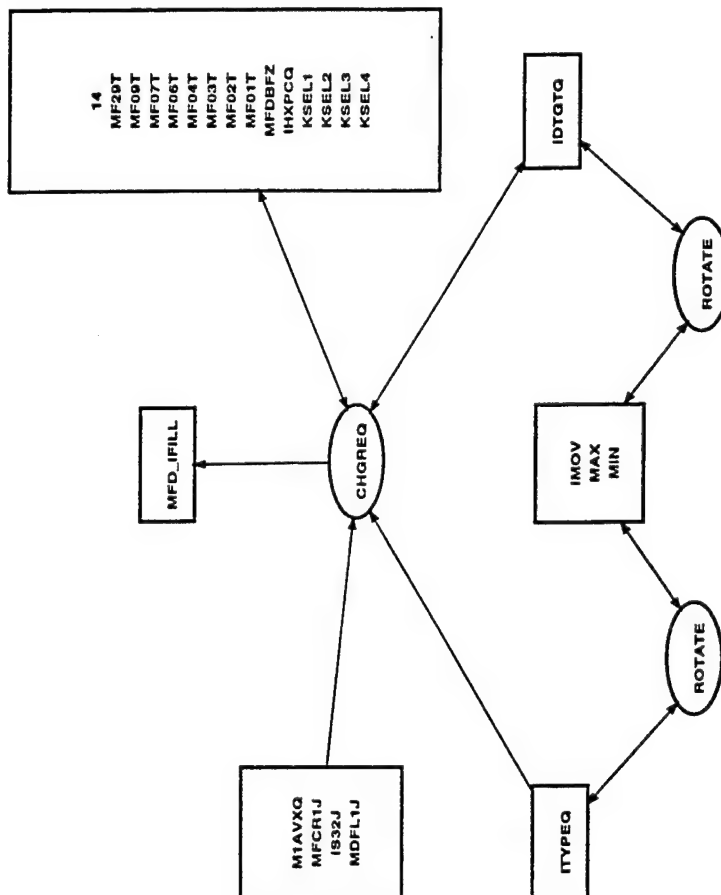




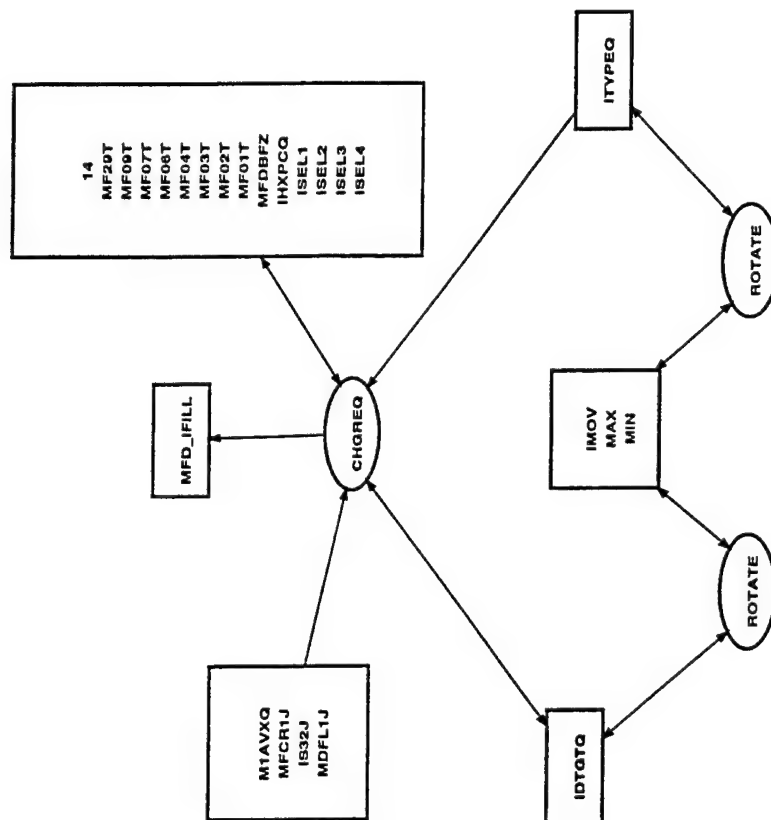






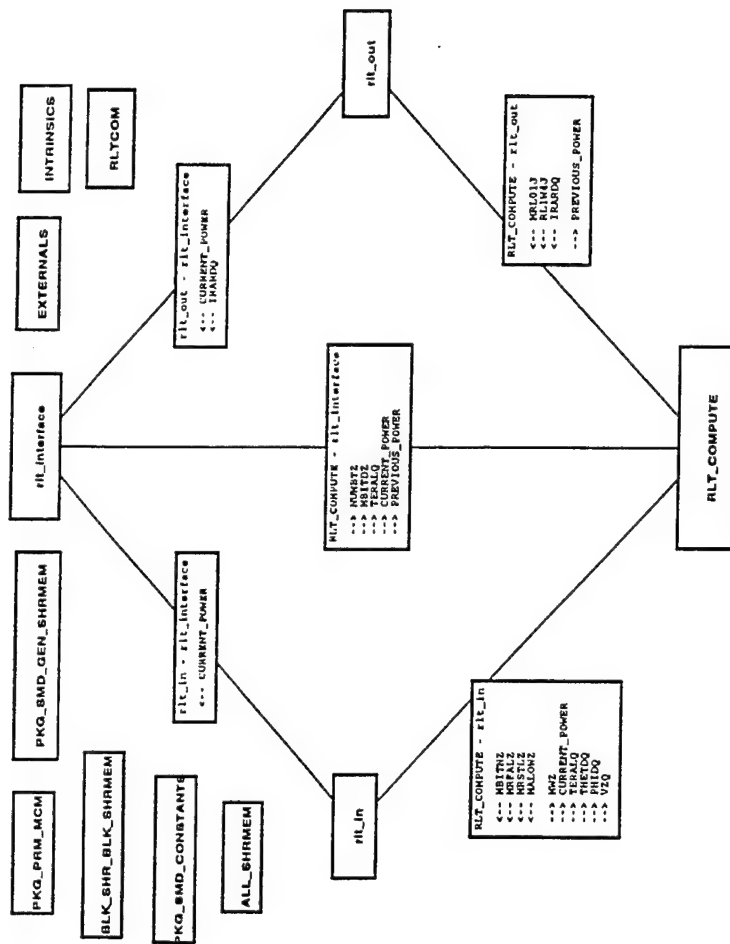


Data Flow Diagram - CHGREQ



APPENDIX I

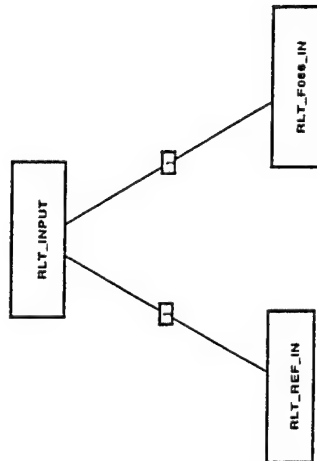
EXHIBIT RLT-P RLT Subsystem Packager Views

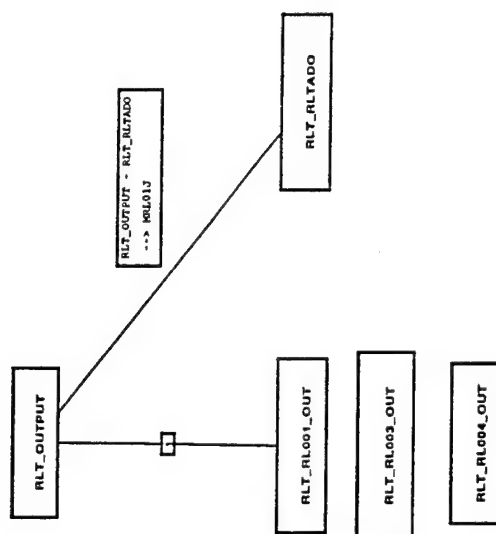


RLT_SUSPEND

RLT_INIT

RLT_TERM





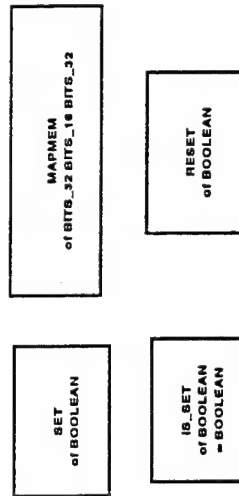
FLOAT4
OF INT_4
REAL_4

JIFIX
OF REAL_4
INT_4

ABS_X
OF REAL_4
REAL_4

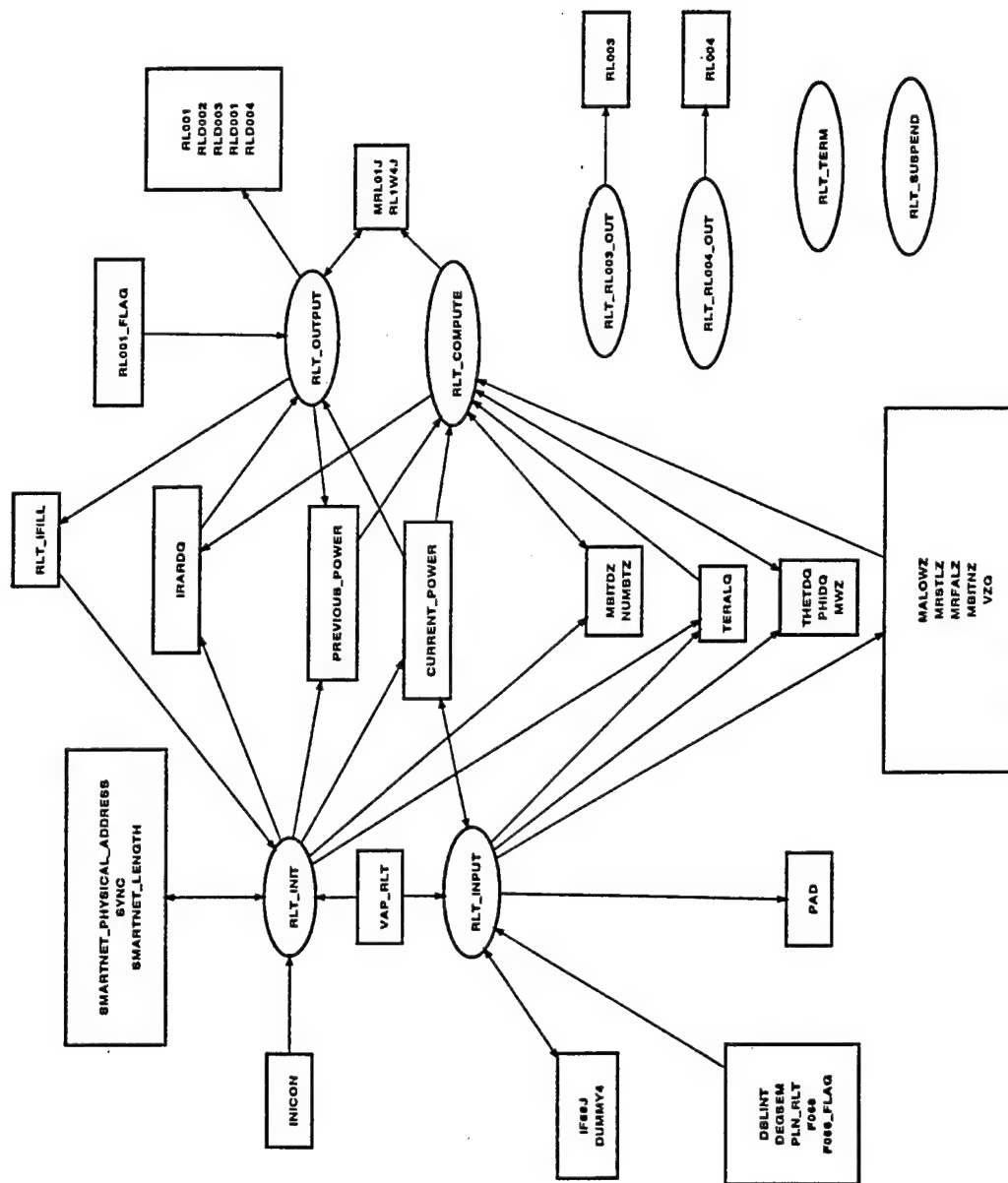
JISHFT
OF INT_4 INT_4
INT_4

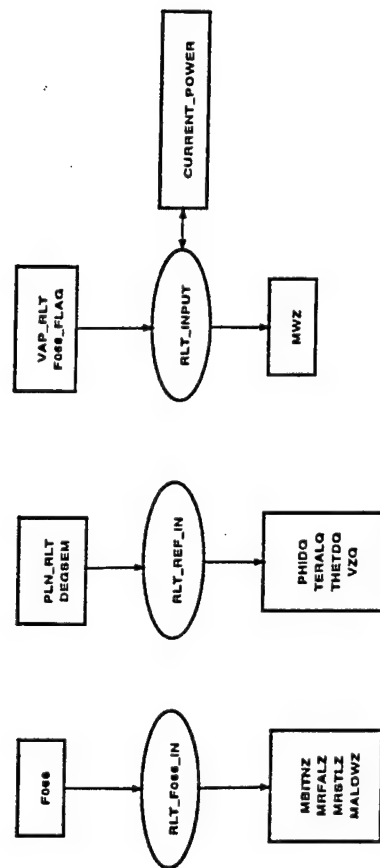
IISHFT
OF INT_2 INT_2
INT_2

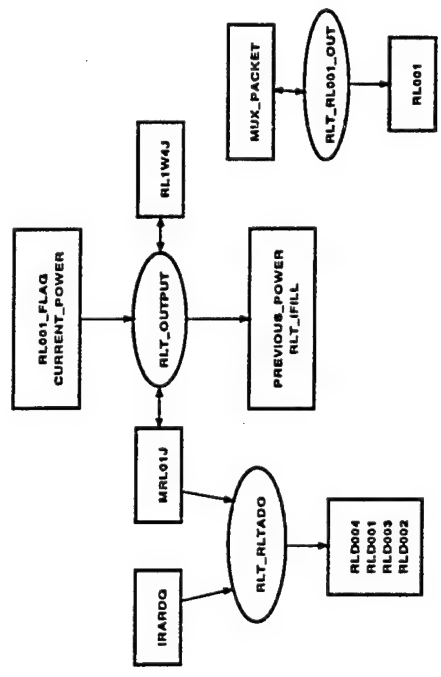


APPENDIX J

EXHIBIT RLT-D RLT Subsystem DFD Views

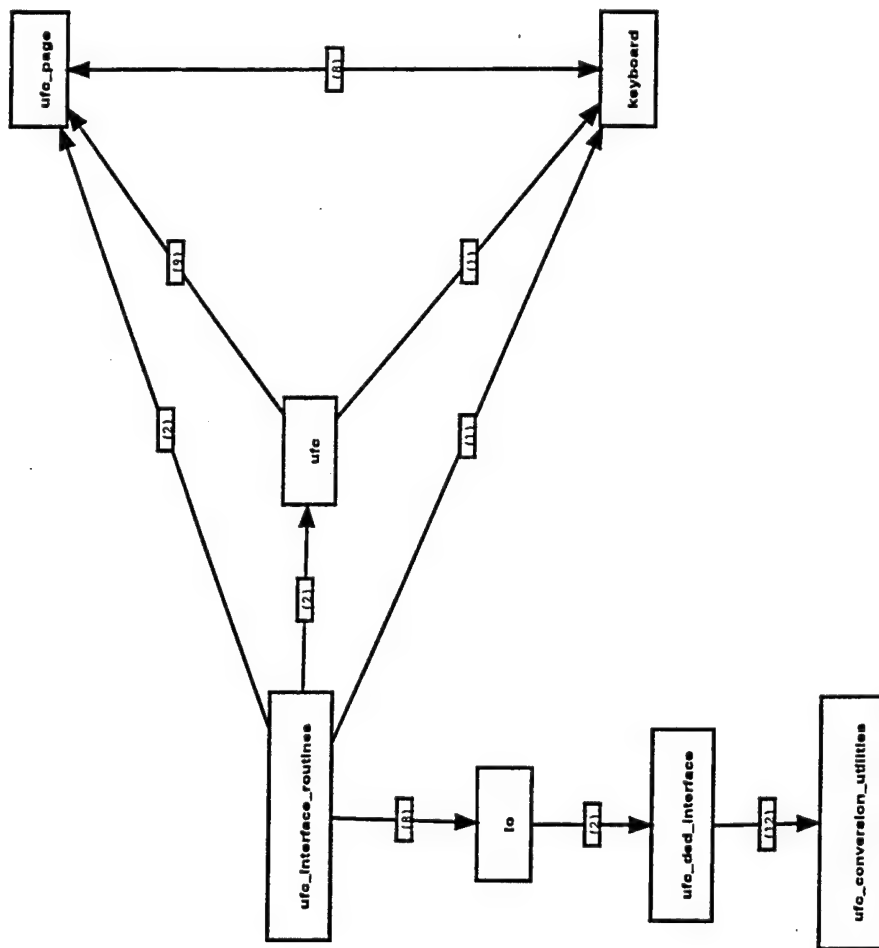


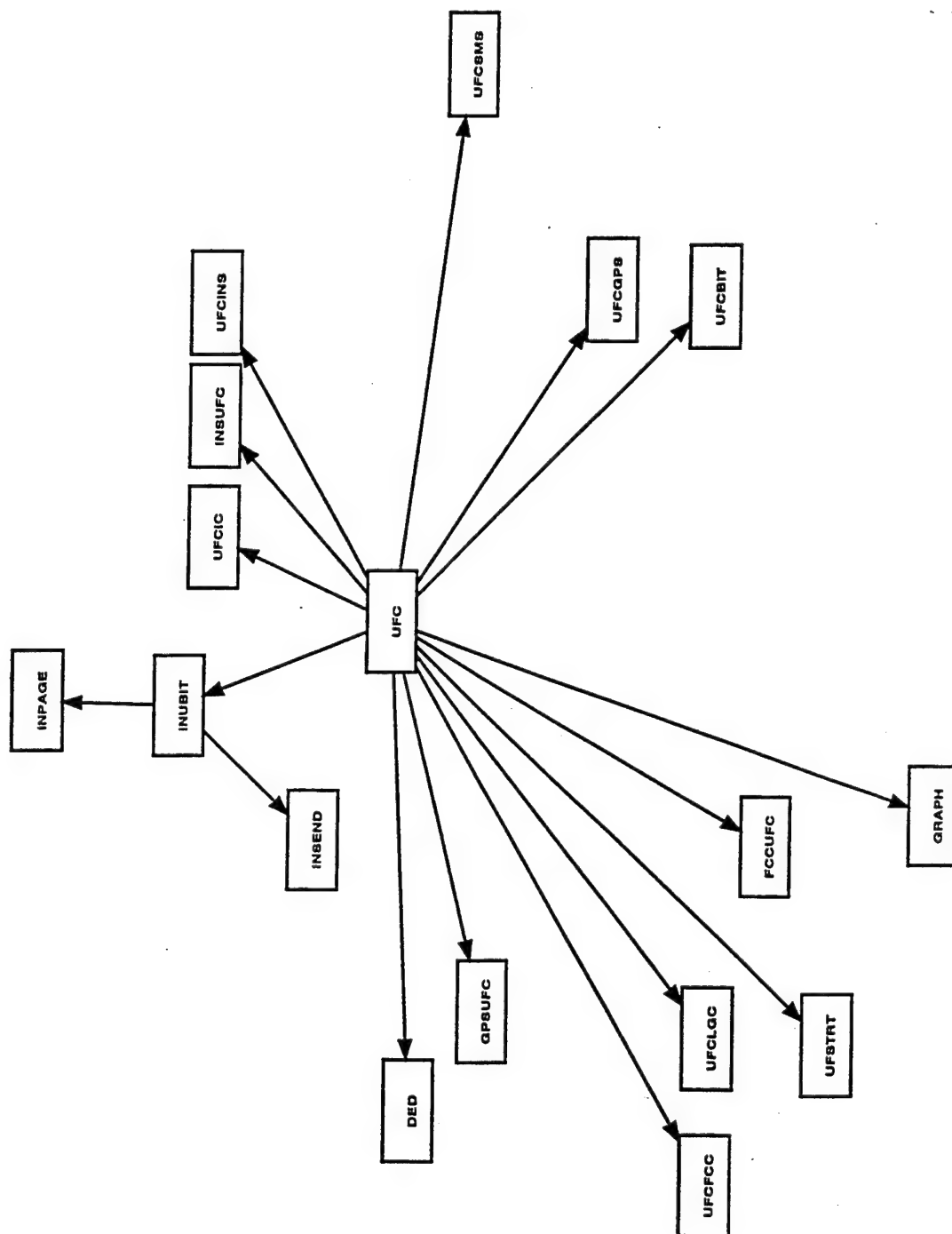


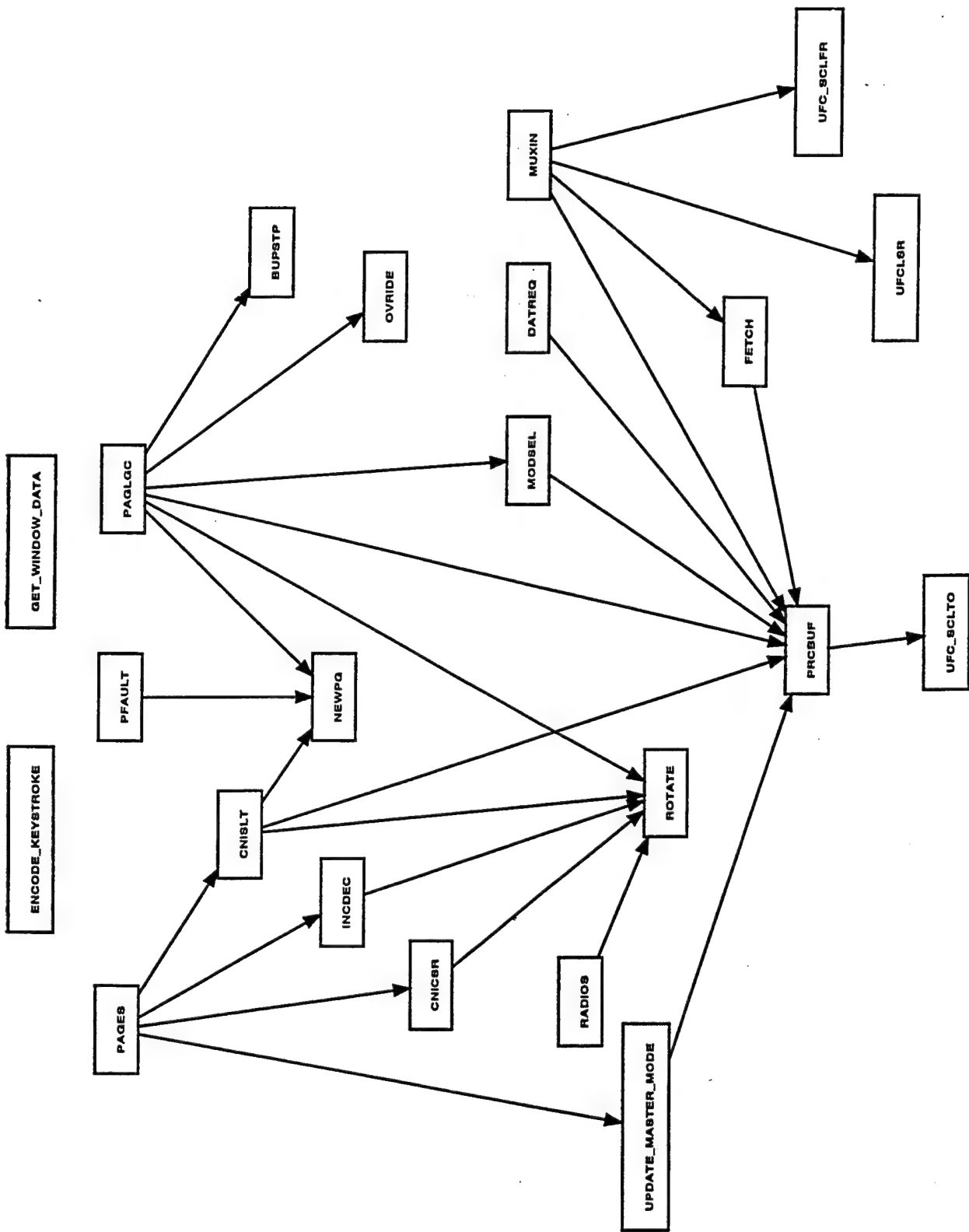


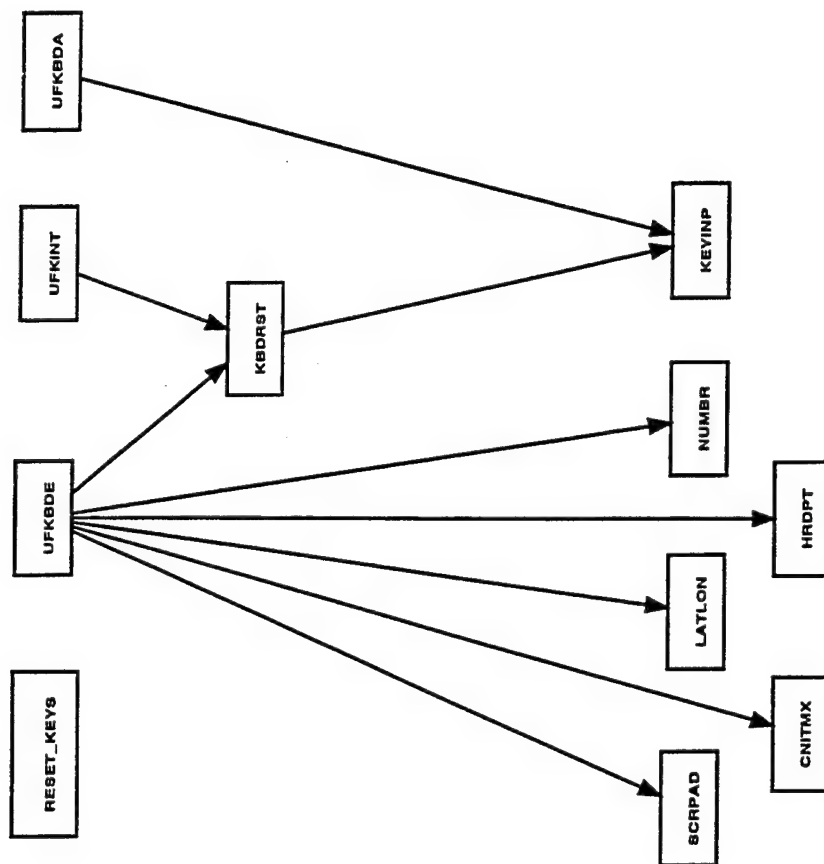
APPENDIX K

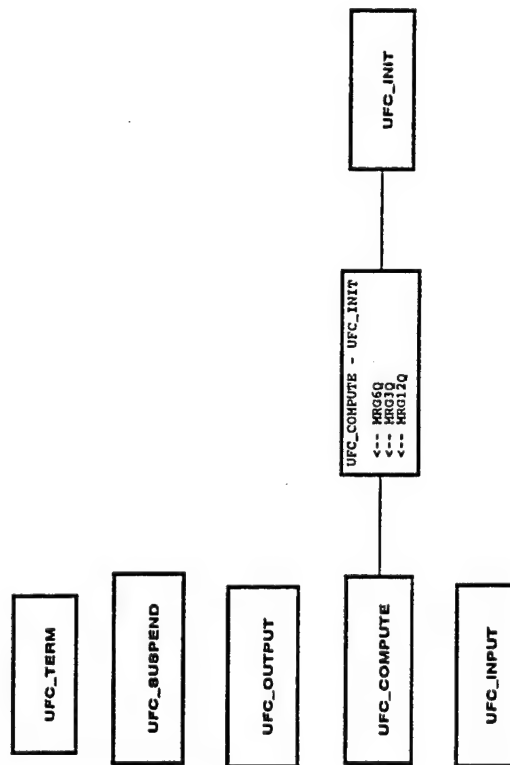
EXHIBIT UFC-P UFC Subsystem Packager Views

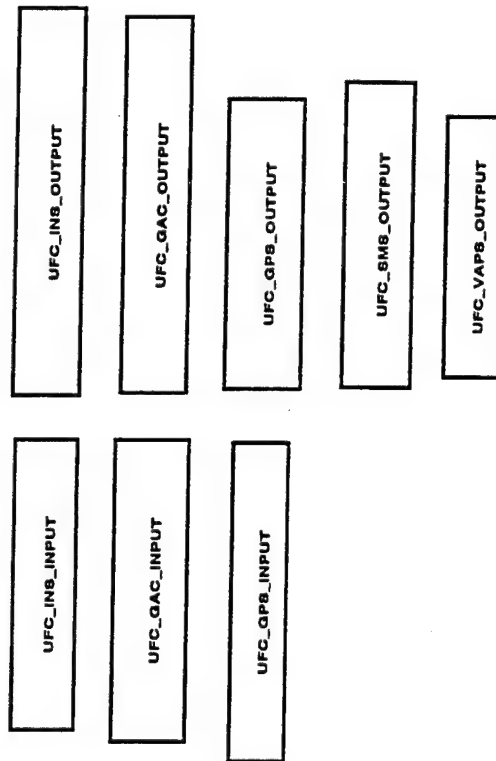






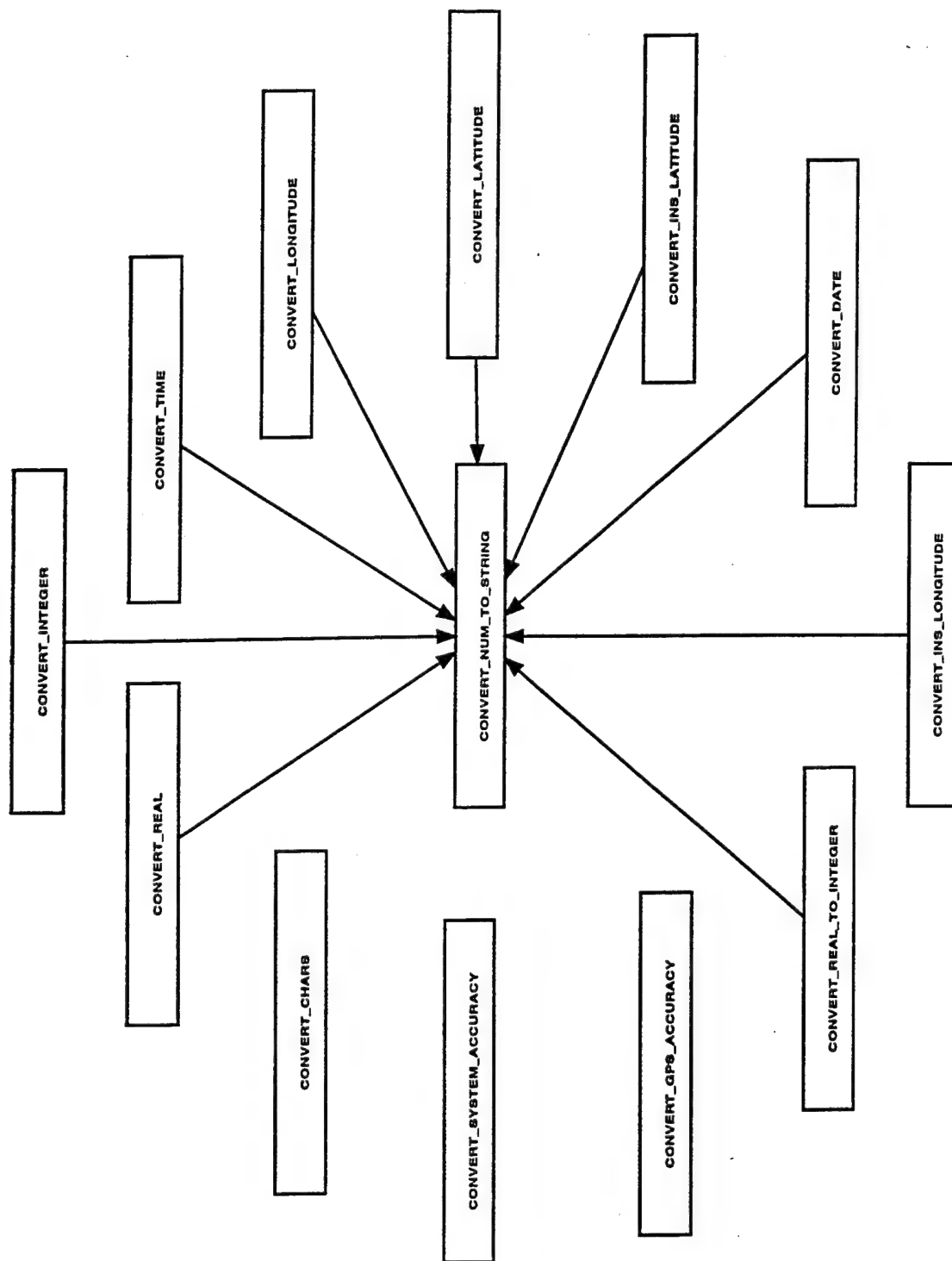






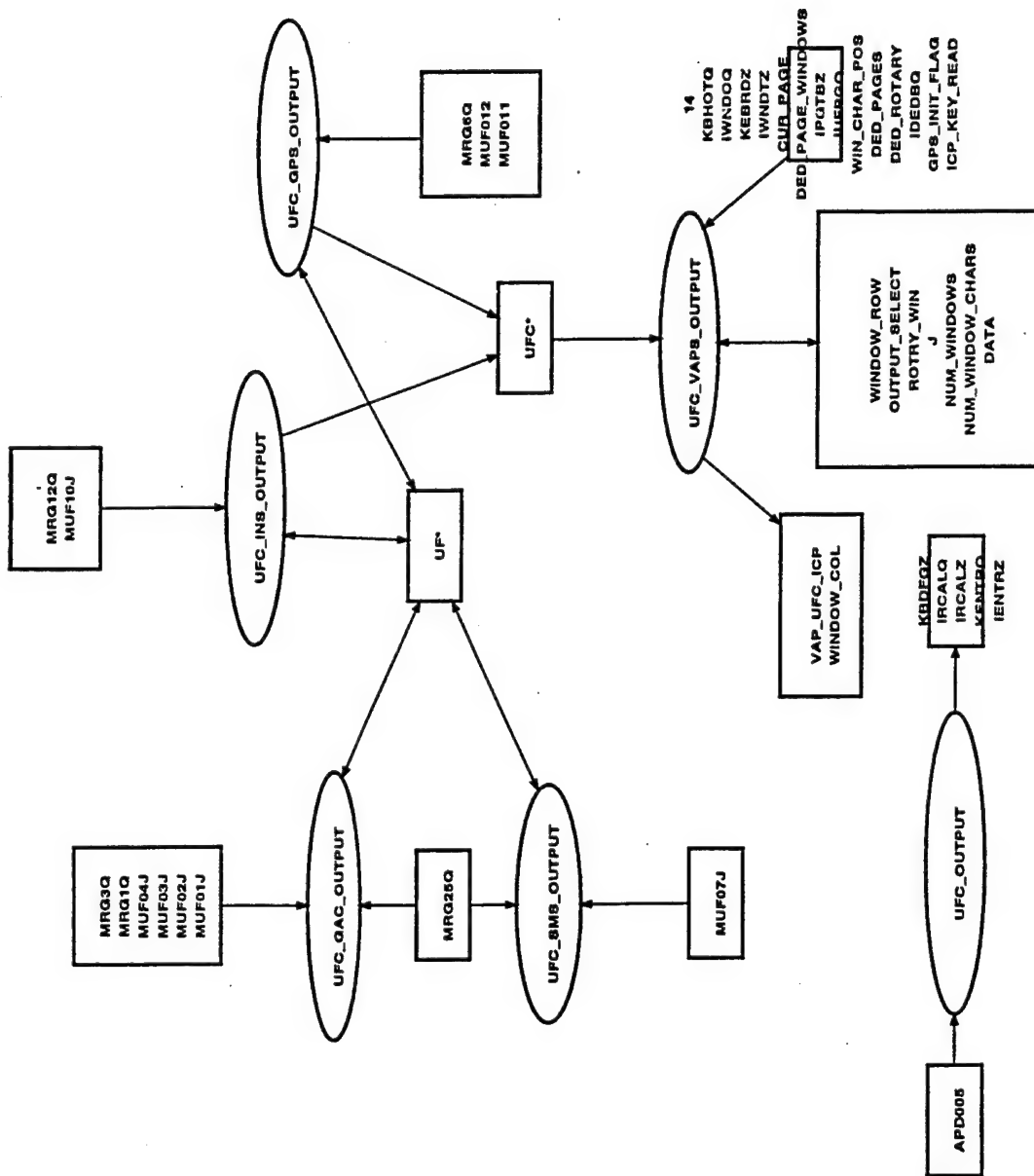
STORE_DYNAMIC_DED_DATA

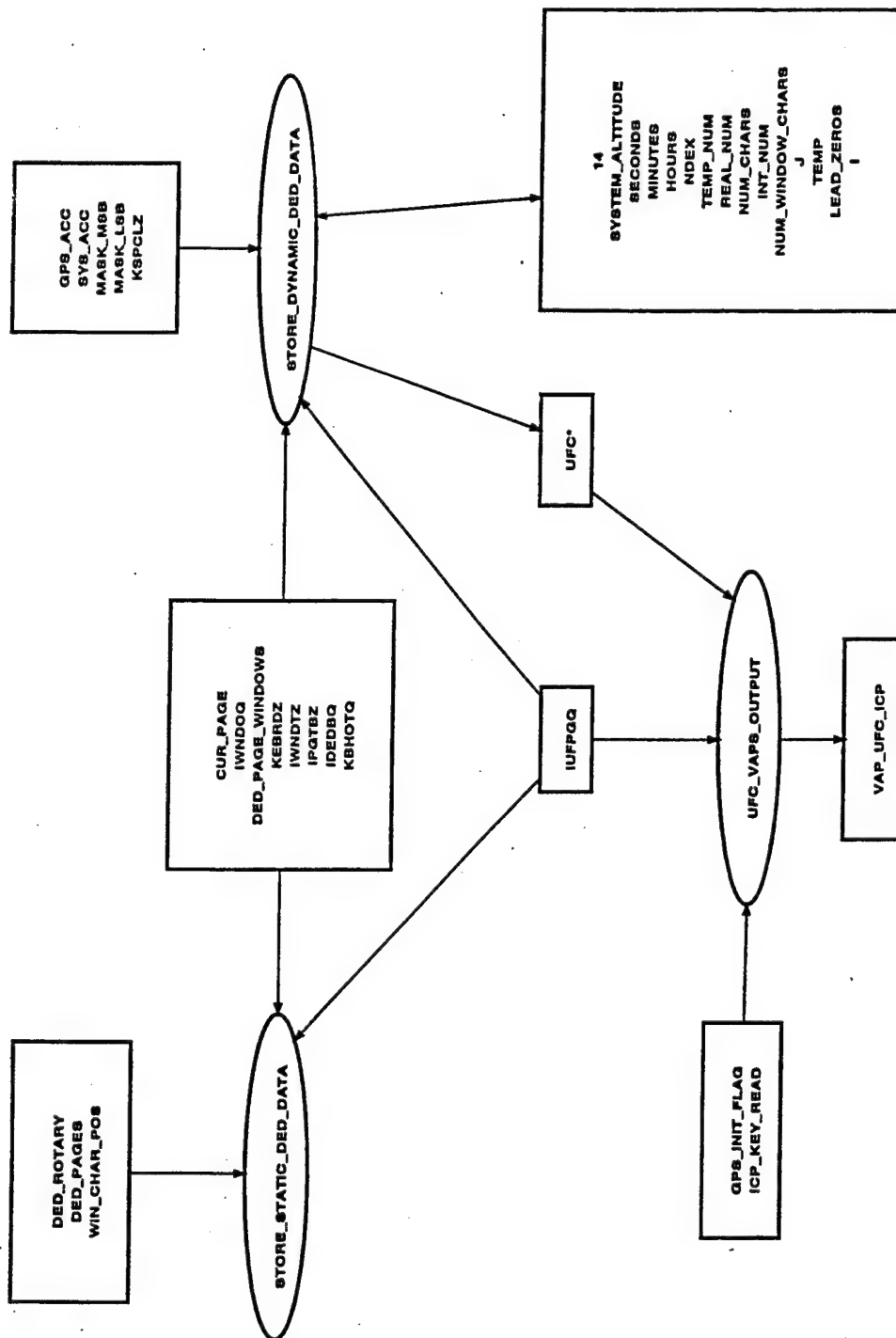
STORE_STATIC_DED_DATA



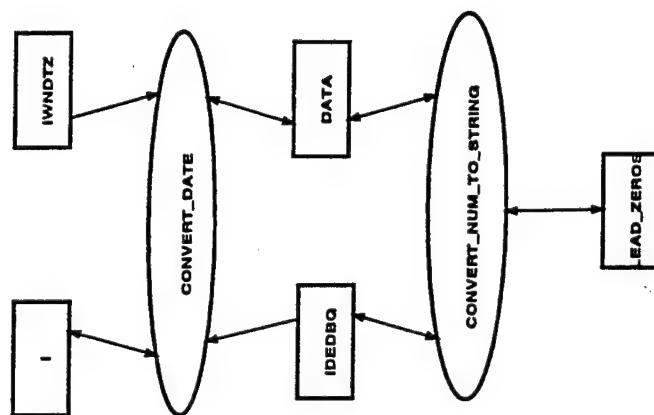
APPENDIX L

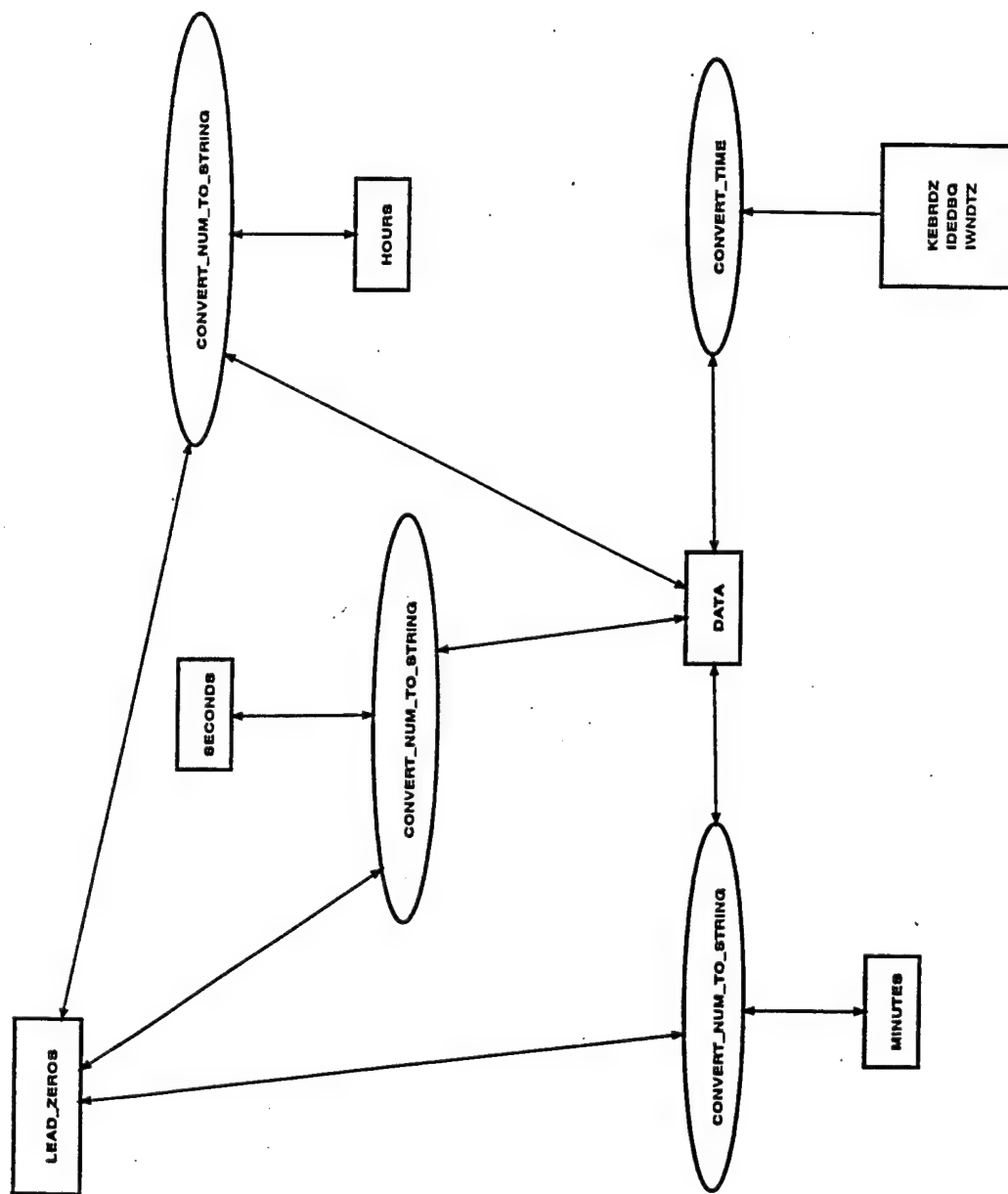
EXHIBIT UFC-D UFC Subsystem DFD Views

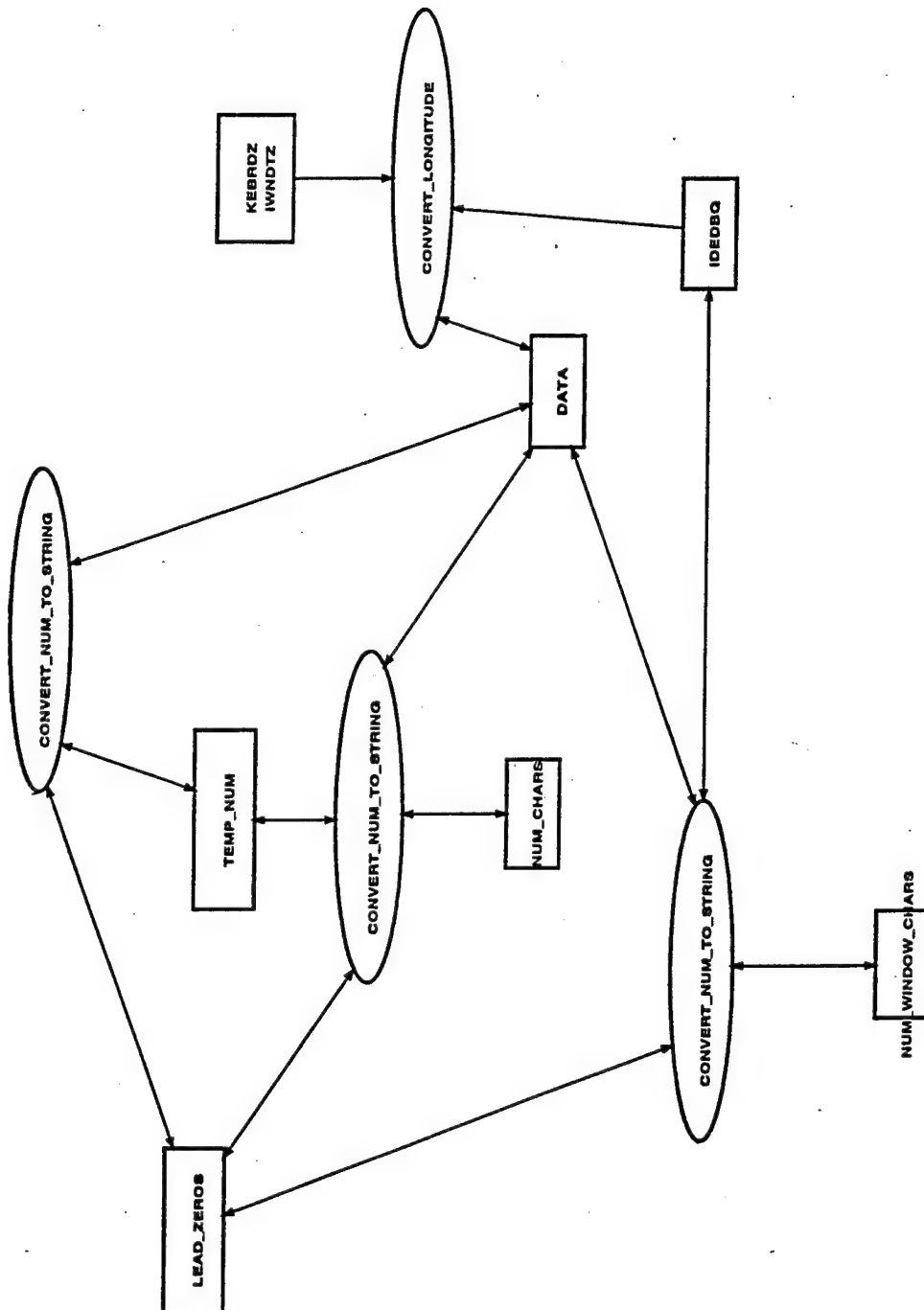


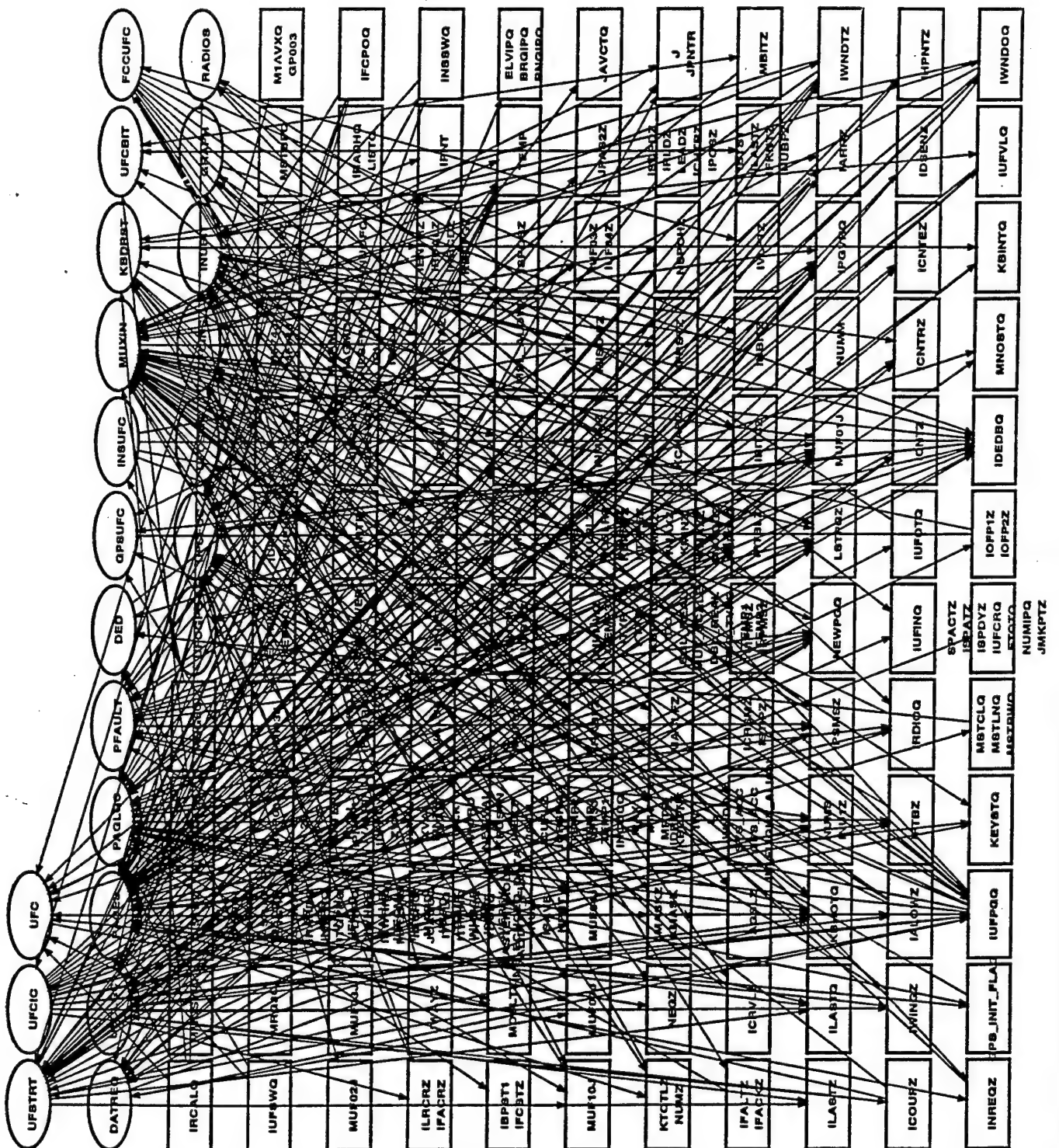


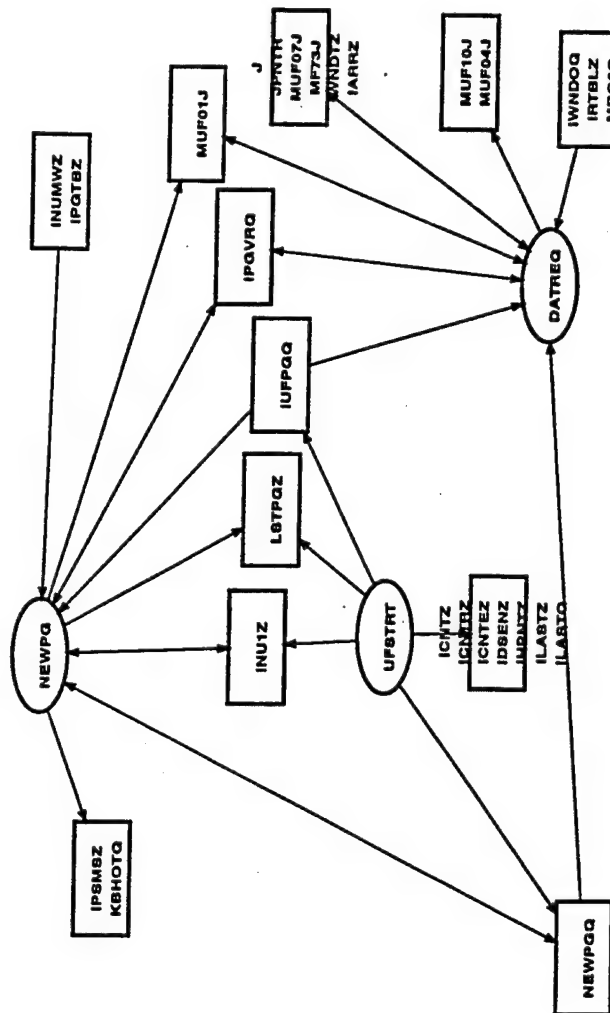
Data Flow Diagram - UFC_VAPS_OUTPUT

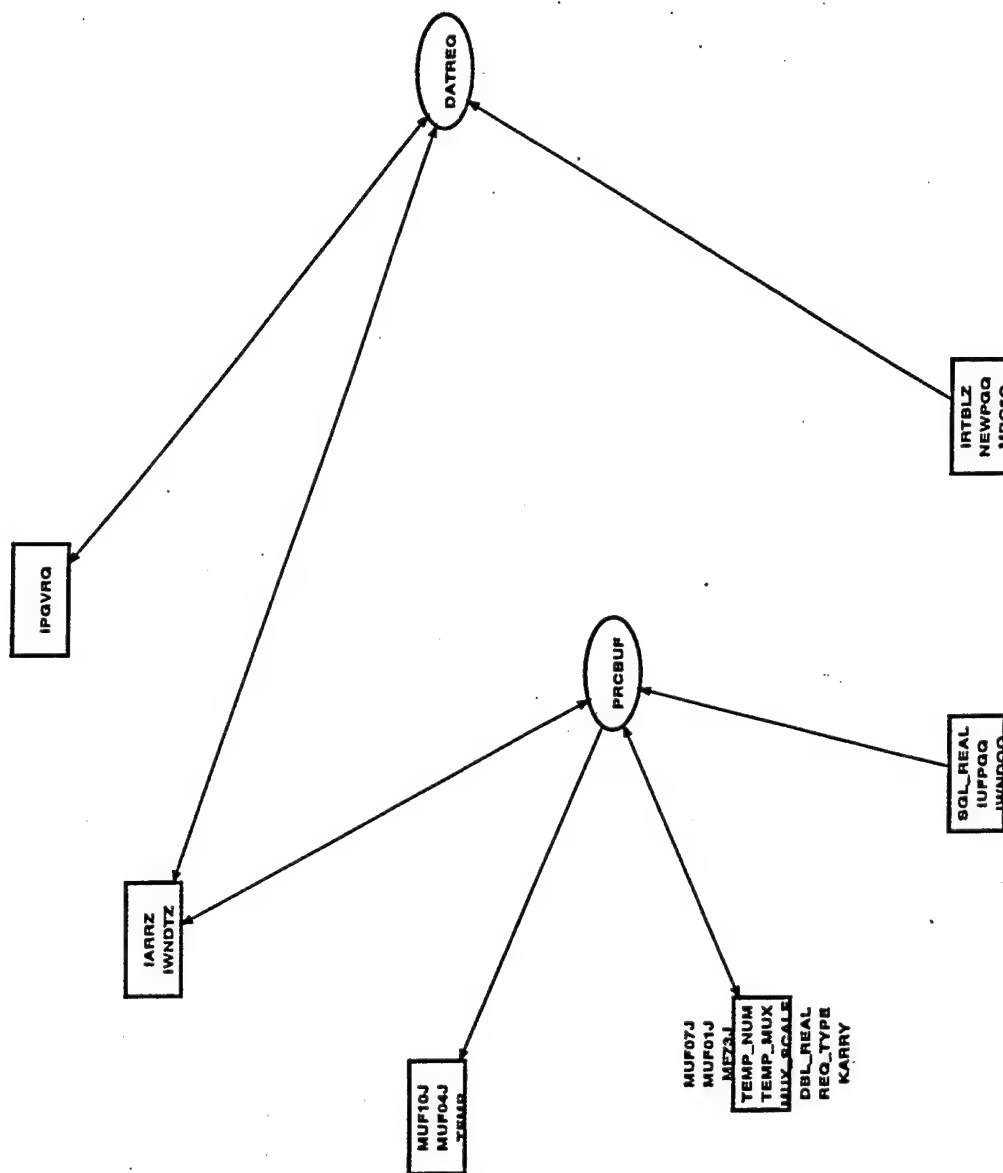




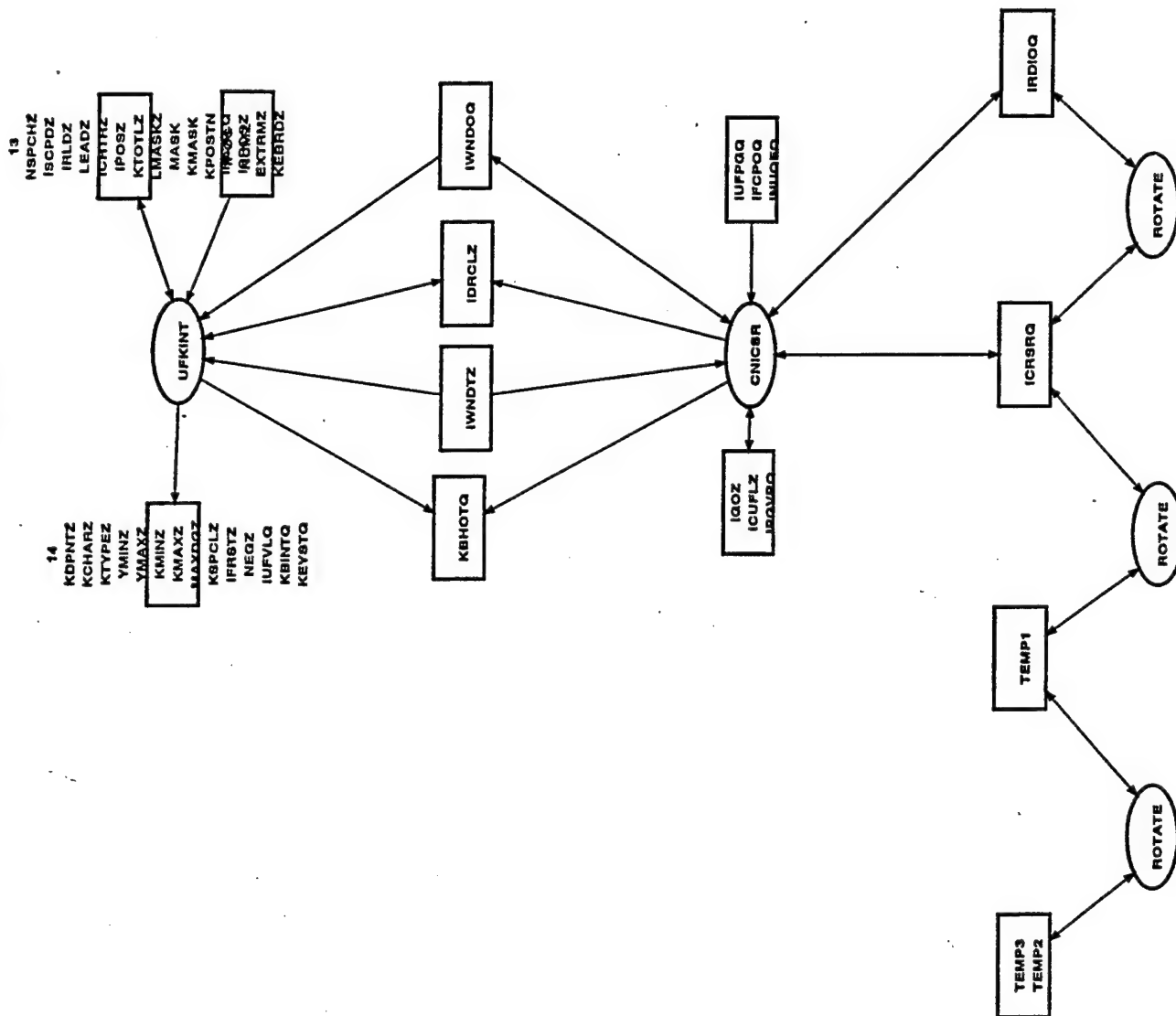


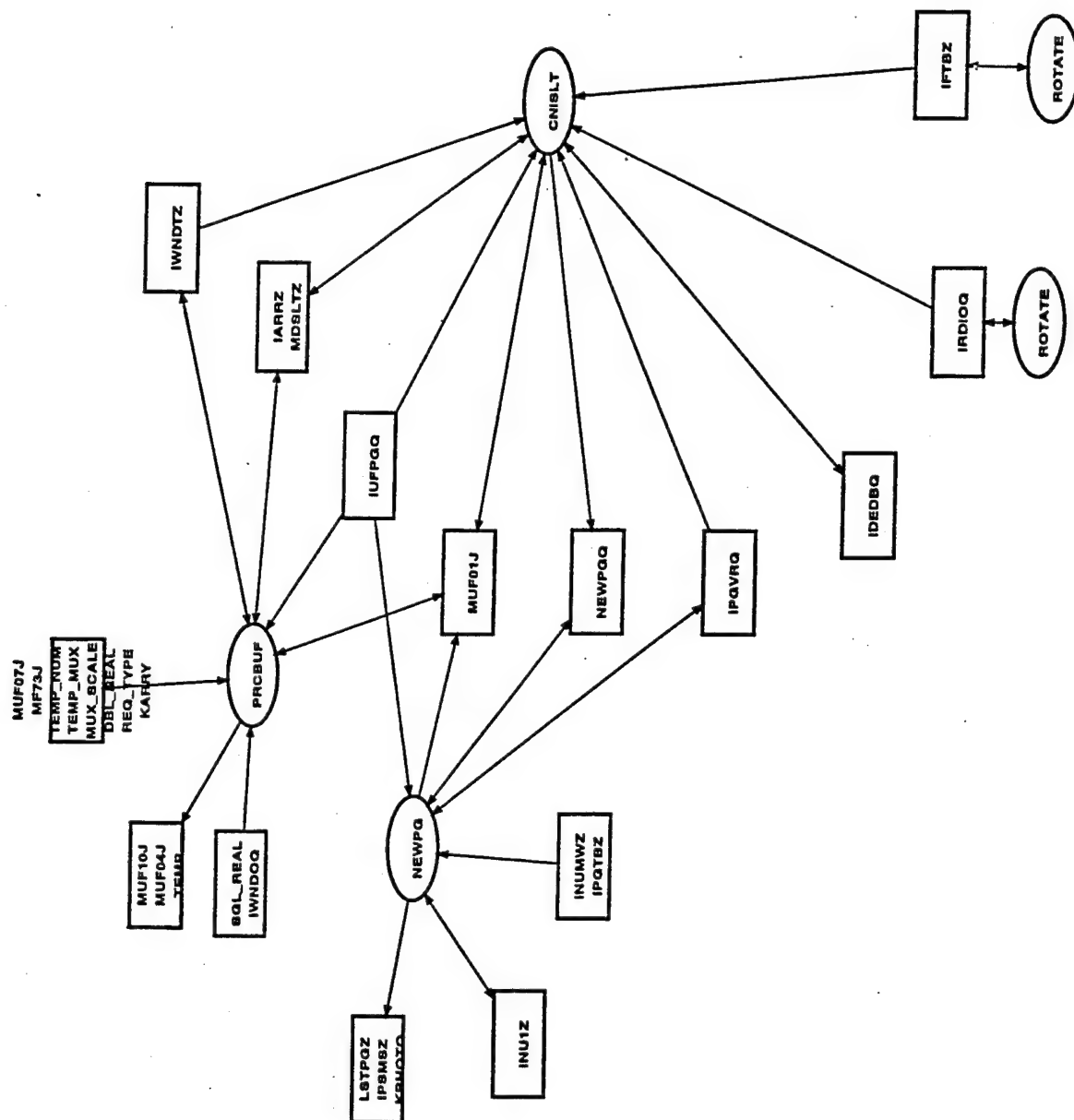


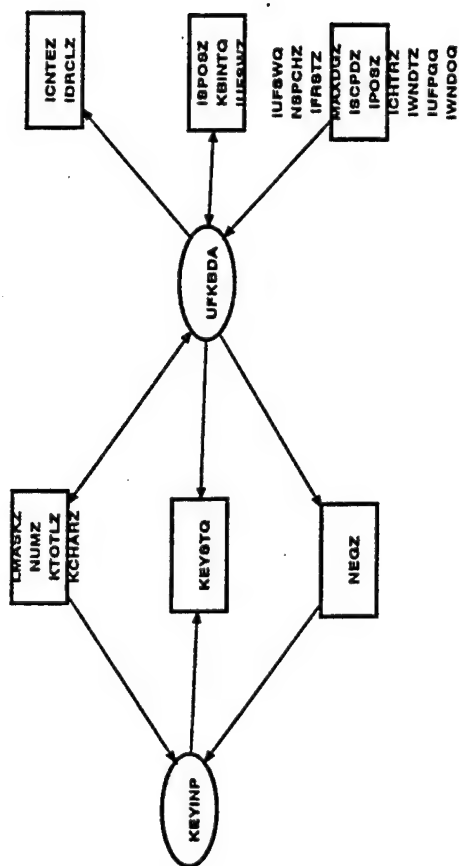


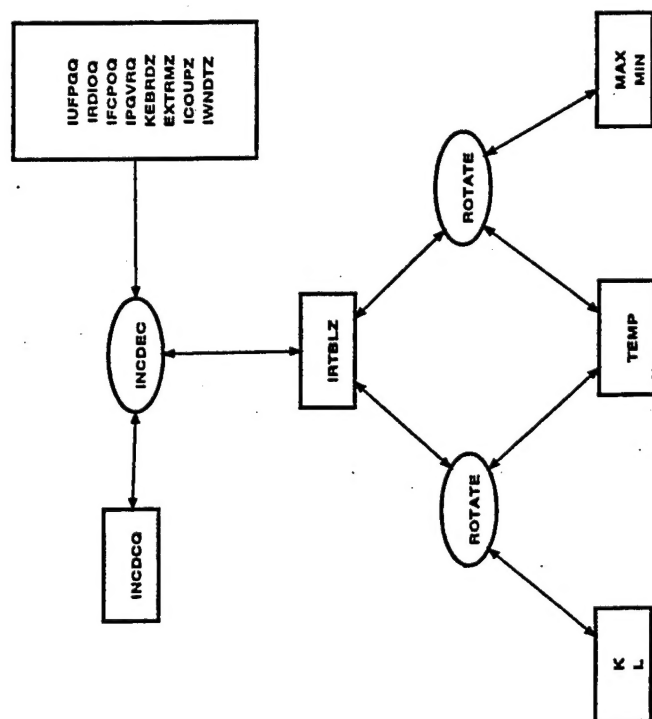


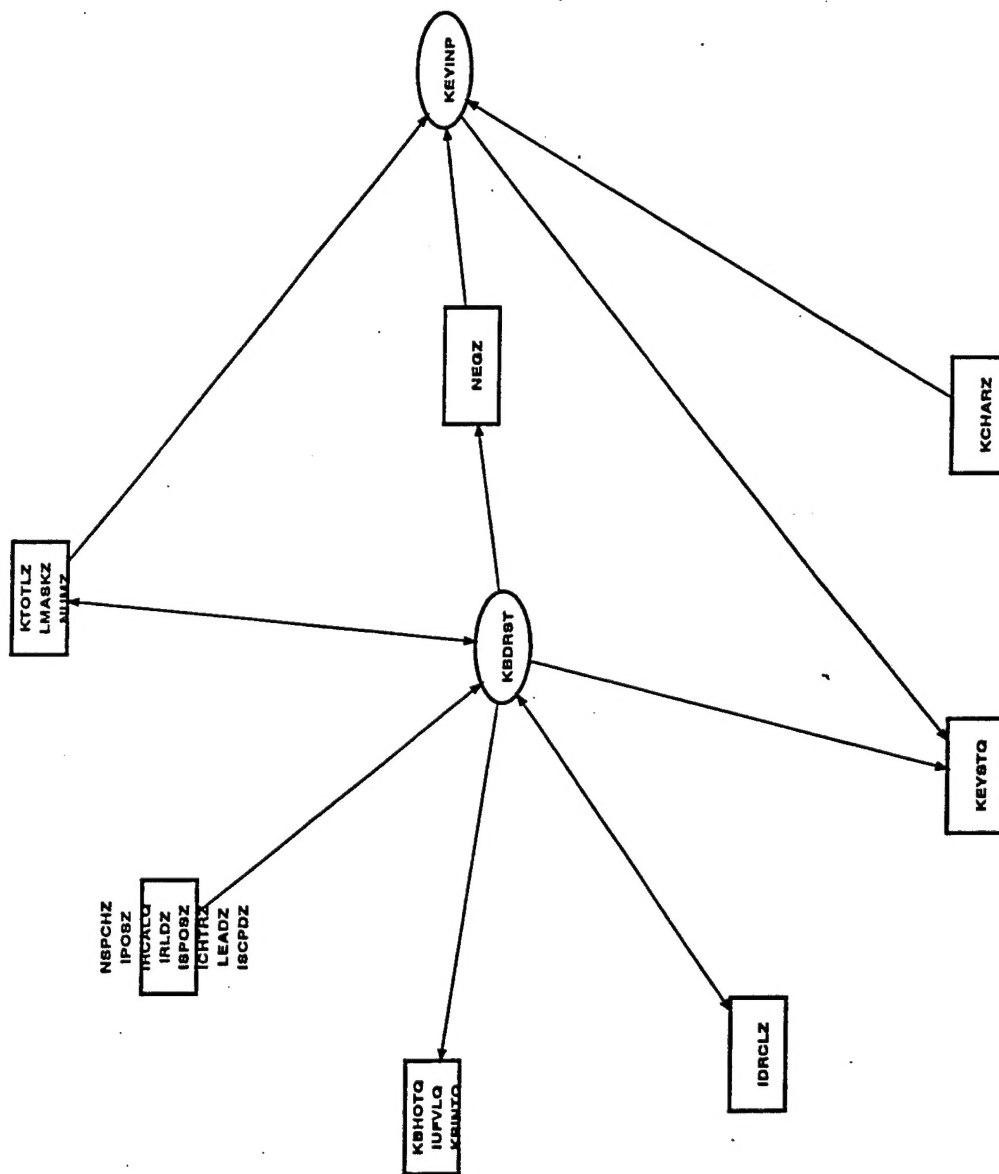












Data Flow Diagram - KBDRST

